RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE
LA RECHERCHE SCIENTIFIQUE

UNIVERSITE MOULOUD MAMMERI DE TIZI-OUZOU
FACULTE DE GENIE ELECTRIQUE ET INFORMATIQUE
DEPARTEMENT D'INFORMATIQUE

Mémoire

En vue de l'obtention du diplôme de

Master en Informatique

Option : Réseaux, Mobilité et Systèmes Embarqués

Thème

Hybridation d'heuristiques pour le problème d'ordonnancement dans les grilles de calcul

Réalisé par :
Mr BOUALI Lyes
Melle BELKADI Halima

Proposé et dirigé par :
Mme OUKFIF

Promotion : 2013-2014

Résumé

Notre objectif est d'implémenter une hybridation de l'heuristique d'ordonnancement HEFT avec une métaheuristique nommée Optimisation par Essaim Particulaire (OEP) dans le but d'optimiser le makespan des applications s'exécutants sur les grilles de calcul.

Pour cela notre mémoire est structuré comme suit :

- Premier chapitre : L'ordonnancement pour les grilles de calcul
  Dans ce chapitre, nous présenterons un état de l'art sur les grilles de calcul et l'ordonnancement sur ce type de plateforme;
- Deuxième chapitre : La métaheuristique PSO
  Ce chapitre présentera la métaheuristique PSO et son fonctionnements;
- Troisième chapitre : L'hybridation HEFT/DPSO
  Au cours de ce chapitre, nous exposerons les techniques d'hybridation entre heuristiques et parlerons sur l'heuristique HEFT et DPSO (version de PSO adaptée pour le problème d'ordonnancement) en détail, puis nous expliquerons la méthode adoptée pour réaliser l'hybridation HEFT/DPSO;
- Quatrième chapitre : Implémentation et Evaluation
  Dans ce chapitre, nous présenterons notre application et ferons une étude comparative de la méthode HEFT/DPSO avec d'autre HEFT, DHEFT et DPSO.

L'étude qu'on a menée dans ce travail montre que les makespan de la méthode que nous avons proposée est meilleur par rapport aux à ceux des autres méthodes. En effet, les makespans de la méthode hybride HEFT/DPSO obtenus pour différentes applications s'exécutant sur différente plateformes sont toujours plus petits par rapport à ceux obtenus avec HEFT et DPSO de base, et plus au moins proches de ceux de DHEFT.

Ce travail nous a permis de nous façonner une idée sur les grilles de calcul, et l'ordonnancement des tâches sur ce types de plate-forme.

Mots clé : Grilles de calcul, Ordonnancement, Makespan, Heuristique, Métaheuristique, HEFT, PSO, Hybridation.

Remerciements

Nous tenons à remercier Dieu tout puissant qui nous a armés de courage, de volonté et surtout de patience.

Nous exprimons notre sincère gratitude à toutes les personnes qui ont rendu ce mémoire possible par leurs aides et leurs contributions.

Nos premiers remerciements sont adressés tout d'abord à notre promotrice Mme OUKFIF pour nous avoir proposé ce sujet et de l'avoir dirigé. Ses conseils et ses encouragements nous ont été précieux, nous apprécions aussi son regard critique sur ce travail.

Nous tenons également à remercier les membres de jury qui ont eu l'amabilité d'examiner et de juger notre mémoire.

Nous gardons une place toute particulière à nos familles, nous leurs exprimons toute notre profonde reconnaissance car ils nous ont constamment aidé, par leur soutien moral et leurs encouragements pour achever ce travail.

Table des matières

Table des matières

Table des matières

Liste des figures

Liste des figures

Liste des tableaux

Depuis quelques années, l'informatique répartie et les technologies associées sont dans une constante évolution technologique et économique. Les technologies sont de plus en plus matures et sophistiquées. Les serveurs de calculs et de stockage voient leur rapport qualité prix en constante augmentation. La même observation peut être appliquée aux technologies réseaux et à la bande passante offerte qui tend à devenir illimitée d'un point de vue applicatif.

La réponse à ces changements serait de passer à un modèle d'informatique répartie permettant d'exploiter pleinement les ressources informatiques disponibles. Cette évolution est connue sous le nom de « Grid Computing » ou « grilles de calculs», en suite on a vu l'apparition du « calcul en nuage » ou le « cloud ».

Cependant l'exploitation de telles infrastructures (grille, cloud) n'est pas toujours facile vu les caractéristique des ressources qui les constituent. Parmi ces difficultés, on cite le déploiement d'une application distribuée, la tolérance aux pannes, le problème de transfert de données et l'ordonnancement des tâches indépendantes et dépendantes sur une grille de calcul. Nous nous somme intéressé à ce dernier problème.

L'objectif de l'ordonnancement est d'assigner les tâches aux ressources disponibles de manière à minimiser le temps de calcul de l'ensemble des tâches (makespan) car l'intérêt de l'utilisateur est d'avoir un programme qui se termine au plus tôt. C'est donc le temps de terminaison de la dernière des tâches qui semble le critère à optimiser le plus pertinent. Pour atteindre cet objectif, plusieurs méthode et technique ont été proposées dans la littérature. Comme le problème d'ordonnancement appartient à la famille des problèmes NP-complet, les métaheuristiques d'optimisations telles que les algorithmes génétiques (GAs) et les algorithmes utilisant les comportements sociaux et évolutifs comme la méthode d'optimisation par essaims particulaires (PSO) peuvent être appliquées pour le résoudre. Ces métaheuristiques offrent des solutions qui peuvent être de qualité mais bien souvent sans garantie.

L'hybridation des métaheuristiques est utilisée avec succès pour résoudre les problèmes difficiles de grande taille. Notre objectif est d'implémenter une méthode d'ordonnancement pour améliorer le makespan des applications dans les grilles de calcul. Nous proposons, dans ce travail, d'hybrider la métaheuristique PSO avec l'algorithme HEFT (Heterogeneous Earliest Finish Time) qui est une heuristique d'ordonnancement très répandu offrant des bons résultats.

Pour cela notre mémoire est structuré comme suit :

Le premier chapitre est composé de deux parties. Nous présenterons dans la première partie un état de l'art sur les grilles de calcul et dans la deuxième partie nous parlerons de l'ordonnancement sur les grilles de calcul;
Le deuxième chapitre présentera quelques métaheuristiques très répandues en s'intéressant à l'Optimisation par Essaim particulaire ou Particle Swarm Optimisation (PSO);

Au cours du troisième chapitre, nous exposerons les techniques d'hybridation. Puis, nous présenterons en détails l'heuristique HEFT et la version discrète de PSO appelée DPSO (Discrete Particle Swarm Optimisation). Ensuite, nous expliquerons notre démarche pour l'hybridation de HEFT et DPSO ;

Dans le dernier chapitre, nous présenterons notre application et effectuons une étude comparative de la méthode HEFT/DPSO avec d'autre HEFT, DHEFT et DPSO. Nous terminons ce mémoire par une conclusion et des perspectives.

Table des matières

1. Introduction

Le développement très dynamique de l'Internet et plus généralement des réseaux de communication a permis la naissance de nouvelles formes d'utilisation de l'informatique durant ces dernières années. Tout d'abord au niveau matériel, ceci a ouvert la porte à la création d'architecture distribué géographiquement [1.1]. Parmi ces architectures distribuées, nous en citons :

- Les grappes de calcul (clusters en anglais) qui désigne un ensemble d'ordinateurs, appelés nœuds, tous inter-connectés, dans le but de partager des ressources informatiques.
- Les grilles informatiques qui désignent un ensemble beaucoup plus important de machines, qui peuvent être réparties sur l'ensemble de la planète et communiquent entre elles en utilisant une grande diversité de réseaux allant de l'Internet à des réseaux privés très haut débit.
- L'informatique en nuage (cloud computing) est une évolution de la notion de grille. Il consiste à proposer des services informatiques sous forme de service à la demande, accessible de n'importe où, n'importe quand et par n'importe qui.[1.2]

Ensuite au niveau logiciel ceci a offert l'opportunité de concevoir de nouveaux modes d'utilisation. Et pour que ces architectures soient utiles, les applications exécutées dessus doivent être distribuées. Ces dernières sont constituées de multitude de tâches à exécuter dans un certain ordre avec des dépendances entre les tâches. Il est primordial d'utiliser au mieux les ressources. Placer quelle tâche sur quelle machine et déterminer à quelle date elles débutent leur exécution, afin d'obtenir une vitesse d'exécution la plus grande possible pour l'application. C'est ce qu'on appelle *un problème d'ordonnancement*.

Dans ce chapitre, nous nous intéresserons au problème d'ordonnancement dans les grilles de calcul. Donc, au premier lieu, nous présenterons un état de l'art sur les grilles informatiques puis, nous exposerons le problème d'ordonnancement dans ce type d'architectures.

2. Etat de l'art des grilles informatiques

2.1. Historique du terme « grille »

« Gille de calcul » provient du terme anglais « The Grid », il a été introduit pour la première fois aux Etats Unis dans les années 1990 pour décrire une infrastructure de calcul répartie utilisée dans des projets de recherche scientifique et industrielle. Le concept de grille informatique puise son inspiration dans le développement des grilles d'électricité (power grids) au début du XXème siècle [1.3].A cette époque, la révolution ne résidait pas en l'électricité elle-même, mais plutôt en la constitution d'un réseau électrique fournissant aux individus un accès fiable et peu onéreux à l'électricité, au travers d'une interface standard : la prise de courant. Les composants formant le réseau électrique sont hétérogènes, et la complexité induite est totalement masquée à l'utilisateur final. Ainsi, une grille informatique possède les mêmes propriétés d'hétérogénéité des ressources que le réseau électrique, le défi scientifique est d'offrir à l'utilisateur de la grille la même transparence d'utilisation qu'offre la prise électrique à l'utilisateur du réseau électrique. La vision des inventeurs de ce terme est qu'il sera possible, à terme, de se « brancher » sur une grille informatique pour obtenir de la puissance de calcul et/ou de stockage de données sans savoir ni où ni comment cette puissance est fournie, à l'image de ce qui se passe pour l'électricité.[1.1] [1.4]

2.2. Définition d'un site

Un site est un ensemble de ressources informatiques localisées géographiquement dans une même organisation (campus universitaire, centre de calcul, entreprise ou chez un individu) et qui forment un domaine d'administration autonome, uniforme et coordonné.

Les ressources informatiques sont aussi bien des liens réseau (câbles, routeurs ou switchs), des machines (simples PC ou calculateurs parallèles) ou des éléments logiciels. [1.5]

2.3. Définition d'une grille informatique

Une grille est un ensemble de ressources géographiquement éloignées sur plusieurs sites, partagées, distribuées, hétérogènes, délocalisées et autonomes et étant organisées virtuellement (les relations entre les entités qui la composent n'existent pas sur le plan matériel mais d'un point de vue logique), afin de réaliser des calculs impossibles à effectue sur un seul site.

Il est important de noter que les grilles informatiques sont, par nature, dynamiques :

1. les sites peuvent quitter ou rejoindre la grille à tout moment ;

2. de même, au sein de chaque site, de nouvelles ressources peuvent être ajoutées, d'autres peuvent tomber en panne ou être déconnectées. [1.5]

2.4. Les objectifs des grilles

L'utilisation des grilles a plusieurs objectifs [1.6] :

- Exploiter les ressources sous-utilisées : Dans la plupart des organisations, il existe une quantité très importante de ressources sous-utilisées. Des études montrent que le taux d'utilisation d'un ordinateur de bureau n'atteint pas les 5 % de moyenne. Ainsi, il est intéressant de pouvoir utiliser ces ressources libres pour exécuter une application lorsque les machines qui lui sont normalement dédiées ne peuvent le faire dans de bonnes conditions, en cas de pics d'utilisation par exemple.
- Fournir une importante capacité de calcul parallèle : Le principal intérêt d'une grille est de permettre l'exécution de plusieurs tâches en parallèle. Ainsi, une application pouvant être découpée en plusieurs tâches, pourra être exécutée sur plusieurs machines de la grille, réduisant ainsi le temps de réponse du calcul à effectuer.
- Accéder à des ressources additionnelles : Outre les processeurs et les capacités de stockage, l'utilisation d'une grille peut s'avérer utile pour accéder à d'autres types de ressources, tels que des équipements spéciaux, des logiciels, et bien d'autres services. Certaines machines peuvent, par exemple, héberger des logiciels ayant des coûts de licence très élevés.
- Mieux répartir l'utilisation des ressources : Puisqu'une grille de calcul permet d'exécuter des applications sur des machines inactives, il est possible de répartir des pics d'utilisation inattendus de certaines machines vers d'autres qui sont moins sollicitées.
- Gérer des applications avec des deadlines proches (délais courts): Si une application doit être exécutée avec une contrainte de date butoir (limite à ne pas dépasser) très proche, l'utilisation d'une grille peut s'avérer utile. En effet, si l'application peut être découpée en un nombre suffisant de tâches, et si une quantité adéquate de ressources peut lui être dédiée, l'application bénéficiera d'une capacité de calcul suffisante pour être exécutée tout en respectant un deadline proche.
- Assurer une tolérance aux fautes pour un coût moindre : Dans les systèmes conventionnels, la tolérance aux fautes est réalisée grâce à la redondance du matériel

sensible. Cette solution possède l'inconvénient d'avoir un coût assez élevé. Les grilles de calcul, de par leur nature, offrent une solution alternative pour effectuer de la tolérance aux fautes. En effet, si une défaillance apparaît sur un site de la grille, les autres parties de la grille ne seront pas forcément affectées. Ainsi, des données peuvent être dupliquées sur plusieurs machines de la grille pour prévenir leur perte en cas de défaillance. De plus, pour des applications temps réel critiques, il peut s'avérer utile d'en exécuter plusieurs instances simultanément sur différentes machines, voire même de vérifier les résultats qu'elles fournissent pour plus de sûreté.

2.5. Caractéristiques des grilles

Les grilles de calcul possèdent quatre principales caractéristiques qui sont [1.4]:

- Multiplicité des domaines d'administration : les ressources sont géographiquement distribuées et appartiennent à différentes organisations chacune ayant ses propres politiques de gestion et de sécurité. Ainsi, il est indispensable de respecter les politiques de chacune de ces organisations.
- Hétérogénéité des ressources : les ressources dans une grille sont de nature hétérogène en terme de matériels et de logiciels. Cette hétérogénéité impose des contraintes de portage de code, d'utilisation de langages multi-plateformes et d'utilisation de protocoles de communication standardisés. L'utilisation de la grille doit être transparente et homogène quelle que soit l'architectures utilisées. [1.1]
- Passage à l'échelle (scalability) : une grille pourra être constituée de quelques dizaines de ressources à des millions voire des dizaines de millions. Cela pose de nouvelles contraintes sur les applications et les algorithmes de gestion des ressources.
- Nature dynamique des ressources : les grilles sont caractérisées par leur aspect dynamique (Arrivée de nouveaux membre, départ des membres existants,…etc.). Cela pose des contraintes sur les applications telles que l'adaptation au changement dynamique du nombre de ressources, la tolérance aux pannes et aux délais…etc.

2.6. Types de grilles

Plusieurs types de grilles peuvent être discernés selon l'utilisation recherchée :

1. Grille d'information (Information Grid): la ressource partagée est la connaissance. L'Internet en est le meilleur exemple : un grand nombre de machines hétérogènes réparties sur toute la surface du globe autorisant un accès transparent à l'information.
2. Grille de stockage (Desktop Grid): l'objectif de ces grilles est de mettre à disposition un grand nombre de ressources de stockage d'information afin de réaliser l'équivalent d'un "super disque dur" de plusieurs PetaBytes. Le projet DataGrid ou les réseaux Kaaza ou Gnutella sont un bon exemple de grille de stockage.
3. Grille de calcul ou grille de serveur (Server Grid) : l'objectif de ces grilles est clairement d'agréger la puissance de traitement de chaque nœud de la grille afin d'offrir une puissance de calcul "illimitée". Notre travail se situe plutôt pour ce type de grille. [1.1]
4. Grille globale (Global Grid) :Pour désigner une grille dont l'ensemble des ressources constitue un supercalculateur virtuel sur lequel l'utilisateur soumet ses applications utilisant des bibliothèques de fonctions spécifiquement conçues. [1.5]

2.7. Les topologies des grilles

[1.4] énumère les grilles d'un point de vue topologique en trois types par ordre croissant d'étendue géographique et de complexité: intragrilles (intragrids), extragrilles (extragrids) et intergrilles (intergrids).

1. Intragrille (en analogie avec Intranet) : la plus simple des grilles est l'intragrille, composée d'un ensemble relativement limitée de ressources et de services et appartenant à une organisation unique. Les principales caractéristiques d'une telle grille sont l'interconnexion à travers un réseau performant et haut débit, un domaine de sécurité unique et maîtrisé par les administrateurs de l'organisation et un ensemble relativement statique et homogène de ressources.
2. Extragrille (en analogie avec Extranet) : une extragrille étend le modèle en regroupant plusieurs intragrilles. Les principales caractéristiques d'une telle grille sont la présence d'un réseau d'interconnexion hétérogène haut et bas débit (LAN/WAN), de plusieurs domaines de sécurité distincts, et d'un ensemble plus ou moins dynamique de ressources. Un exemple d'utilisation est lors d'alliances et d'échanges « Business -toBusiness » (B2B) entre entreprises partenaires.

3. Intergrille (en analogie avec Internet) : une intergrille consiste à agréger les grilles de multiples organisations, en une seule grille. Les principales caractéristiques d'une telle grille sont la présence d'un réseau d'interconnexion très hétérogène haut et bas débit (LAN / WAN), de plusieurs domaines de sécurité distincts et ayant parfois des politiques de sécurité différentes et même contradictoires, et d'un ensemble très dynamique de ressources.

2.8. Avantages et inconvénients des grilles

Les grilles présentent plusieurs avantages et inconvénients que nous citons [1.5] :

2.8.1 Avantages

- Exploiter les ressources sous utilisées : Les études montrent que les ordinateurs personnels et les stations de travail restent inactifs la plupart du temps. Les grilles de calcul permettent ainsi d'utiliser les cycles processeurs durant lesquels les machines sont inactives afin d'exécuter une application ou une partie d'application nécessitant une puissance de calcul importante.
- Fournir un cadre distribué : Une grille de calcul peut agréger une importante quantité de ressources afin de fournir une puissance de calcul aussi performante que les gros calculateurs parallèles. Les ressources agrégées peuvent aller du simple PC à des calculateurs parallèles.
- Fiabilité et disponibilité des services : Les ressources fédérées par une grille de calcul sont géographiquement éloignées et disponibles en importantes quantités. Cela permet d'assurer la continuité du service si certaines ressources deviennent indisponibles. Dans ce cas, les logiciels de contrôle et de gestion de la grille sont en mesure d'allouer d'autres ressources et de leur transmettre les calculs à effectuer.
- Gestion adéquate des ressources : En partageant les ressources, une grille peut fournir l'accès à des ressources spéciales comme des équipements (bras robotiques, cameras, ...) ou des bibliothèques (BLAS, LAPACK). Ainsi ces ressources seront utilisées et partagées par plusieurs utilisateurs.

2.8.2. Inconvénients

- L'architecture de la grille est caractérisée par une grande hétérogénéité des ressources et par l'absence d'une connaissance à priori des ressources mises à disposition.

- Le grand nombre de ressources et leur partage rend difficile leur gestion.
- L'adaptation des applications existantes au calcul distribué qui est souvent un travail lourd et fastidieux.
- Plusieurs problèmes peuvent apparaitre du faite de la répartition des sites d'une grille comme la sécurité, l'harmonisation des procédures de sauvegarde, le temps de transfert des données et la qualité de service sur les liaisons longues distances…etc.

2.9. L'architecture générale d'une grille

La figure suivante illustre l'architecture générale d'une grille informatique [1.7].

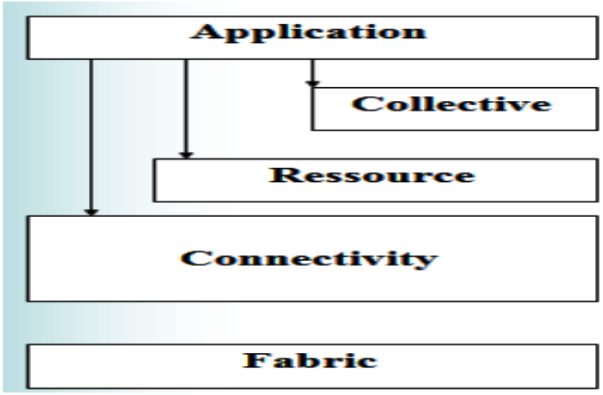


Figure 1.1 : Les couches d'une grille

1. **La couche Fabrique**: Elle fournit les ressources. Ce sont d'un point de vue physique des ressources telles que des processeurs pour le calcul, du stockage, des bases de données, des annuaires ou des ressources réseau. Une ressource peut-être également une entité logique comme un système de fichiers distribué, ou un serveur virtuel dans le cas d'un cluster d'ordinateurs. La couche Fabrique est la plus basse du modèle. Elle est en relation directe avec le matériel pour mettre à disposition les ressources partagées.

   Lorsqu'une demande d'accès à une ressource est formulée, par le biais d'une opération de partage d'un niveau supérieur, un composant logiciel du niveau Fabrique est invoqué. Le rôle de ce composant est d'agir directement sur les ressources logiques ou physiques de la grille.

2. **La couche Connectivité**: elle implémente les principaux protocoles de communication et d'authentification nécessaire aux transactions sur un réseau de type grille. Les protocoles de communication permettent l'échange des données à

travers les ressources du niveau Fabrique. Ces protocoles d'authentification s'appuient sur les services de communication pour fournir des mécanismes sécurisés de vérification de l'identité des utilisateurs et des ressources.

Les protocoles nécessaires sont ceux relatifs au transport, au routage et à la gestion des noms. Le choix se porte essentiellement sur les protocoles de la pile TCP/IP. TCP et UDP sont utilisés pour le transport, IP pour le routage, ICMP pour la surveillance, et pour les applications est utilisé DNS.

3. **La couche Ressource**: Utilise les services des couches Connectivité et Fabrique pour collecter des informations sur les caractéristiques des ressources, les surveiller et les contrôler. Elle s'occupe également de l'aspect facturation. La couche Ressource ne se préoccupe pas des ressources d'un point de vue global, elle ne s'intéresse pas à leur interaction. Ceci incombe à la couche Collectif abordée dans la section suivante. La couche Ressource ne s'intéresse qu'aux caractéristiques intrinsèques des ressources et à la façon dont elles se comportent.

4. **La couche Collectif** : Elle se charge des interactions entre les ressources. Elle gère l'ordonnancement et la co-allocation des ressources en cas de demande des utilisateurs faisant appel à plusieurs ressources simultanément. C'est elle qui choisit sur quelle ressource de calcul faire exécuter un traitement en fonction de coûts qu'elle sait estimer. Elle s'occupe également des services de réplication des données. En outre, elle a en charge la surveillance des services et elle doit assumer la détection des pannes. En un mot elle a un rôle « d'orchestrateur » de l'ensemble des ressources du système.

5. **La couche Application**: La couche la plus haute du modèle est la couche Application qui correspond aux logiciels qui utilisent la grille pour fournir aux utilisateurs ce dont ils ont besoin, qu'il s'agisse de calcul, ou de données. Les applications utilisent des services de chacune des couches de l'architecture. Les couches Collectif et Ressources sont sollicitées pour la recherche des ressources. Après les avoir trouvées, et après s'être authentifiées à travers de la couche Connectivité, les applications utilisent les services du niveau Fabrique pour y accéder.

2.10. Les intergiciels des grilles

Pour exploiter une grille, on utilise un ensemble de logiciels appelé intergiciel (middleware). Son rôle est d'unifier l'accès à des ressources informatiques hétérogènes. Il se place entre les systèmes d'exploitation existants et l'utilisateur. Il masque à ce dernier l'hétérogénéité des divers systèmes installés et fournit un ensemble de routines (commandes et bibliothèque de programmation) indépendantes du système. Ces commandes et fonctions permettent par exemple d'exécuter une application sur une machine avec un langage de description de ressources (quantité de mémoire nécessaire, temps CPU) indépendant de la machine utilisée [1.5]

Parmi les intergiciels les plus répondus dans les projets de grille, nous citons Globus [1.8], utilisé par la grille scientifique américaine OGS (Open Grid Science) et la grille européenne EGEE (EGEE utilise maintenant une évolution appelé GLite [1.5]). Nous trouvons aussi le middleware Legion[1.4], UNICORE, et Gridbus [1.8].

Les principales fonctions assurées par ces intergiciels sont [1.9]:

- **Ordonnancement:** L'ordonnanceur doit connaître à tout moment la charge de chaque machine de la grille afin de pouvoir attribuer aux ressources les plus sous exploitées les tâches de calculs. Il supervise aussi du début à la fin le déroulement d'un job. Il peut le soumettre à nouveau s'il est brusquement interrompu et le terminer prématurément s'il se trouve dans une boucle infinie d'exécution.
- **Réservation:** Cette fonction permet de réserver des ressources à l'avance dans le but de garantir une certaine qualité de service et de respecter certaines échéances.
- **Services d'information et d'annuaire (Information and Directory services):** Une grille est un environnement où le nombre et la charge des ressources sont dynamiques. Il est alors nécessaire de fournir des mécanismes permettant d'enregistrer et de découvrir ces ressources tout en identifiant leurs statuts.
- **Service de nom (Naming Service) :** Comme dans tout système réparti, une grille devra permettre de référencer ses ressources d'une façon standard et uniforme. Une grille de calcul devra fonctionner avec un large spectre de technologies matérielles ou logicielles. Tout comme les utilisateurs du WEB ne se soucient pas de savoir si les serveurs WEB tournent sur une machine x86 ou Sparc ou bien si le système d'exploitation est Unix ou Windows, les utilisateurs d'une grille ne veulent pas se

préoccuper des détails matériels et logiciels de l'infrastructure. Un environnement de grille idéal fournira ainsi un accès transparent aux ressources en cachant toutes les différences « physiques ».

2.11. Les domaines d'application

Les principales applications des grilles de calculs se font dans les domaines suivants [1.10]:

- Le calcul haute performance distribué : son but est d'obtenir assez de ressources pour effectuer des calculs fortement dépendant les uns des autres.
- Le calcul haut débit : à la différence du calcul haute performance, un grand nombre de calculs indépendants doivent être résolus.
- Le calcul à la demande : répond à un besoin sporadique (irrégulier) de ressources de calcul ou de stockage.
- Le calcul orienté données, pour lequel l'analyse de données géographiquement distribuées permet la synthèse de nouvelles données.
- Le calcul collaboratif qui propose à des utilisateurs distribués géographiquement un environnement de stockage et de calcul permettant une collaboration.

2.12. Les services fournis par les systèmes de grille de calcul

Les systèmes de grilles de calcul peuvent être employés pour fournir les services suivants [1.11]:

- Services de calcul (Computational Services) : Il s'agit de fournir des services sécurisés pour exécuter les tâches des applications sur les ressources informatiques distribuées. Quelques exemples de grilles informatiques sont NASA IPG, World-Wide Grid, et NSF TeraGrid.
- Services de données (Data Services) : Ils cherchent à permettre le transfert sécurisé aux ensembles de données distribuées et à leur gestion. Un exemple d'application qui a besoin d'un tel service pour la gestion, le partage et le traitement de grands ensembles de données est la Physique des Hautes Energies.
- Services d'application (Application Services) : Ceux-ci ont pour objectif de gérer des applications et de fournir un accès à distance aux logiciels et aux bibliothèques

d'une manière transparente. Un exemple qui peut être employé pour développer de tels services est NetSolve.

- Services d'information (Information Services) : Ceux-ci se spécialisent dans l'extraction et la présentation des données significatives en employant les services données et applications.
- Services de connaissance (Knowledge Services) : Ils se focalisent sur la manière d'acquérir, d'employer, de rechercher, d'éditer, et de maintenir des connaissances pour aider les utilisateurs à atteindre leurs buts et objectifs particuliers.

3. L'ordonnancement sur les grilles informatiques

3.1. Généralités

3.1.1. Problème d'ordonnancement

Le problème d'ordonnancement consiste à organiser dans le temps l'exécution d'un ensemble de tâches, compte tenu des contraintes temporelle tel que le délai et d'autres contraintes portant sur l'utilisation et la disponibilité des ressources requises [1.12], tout en satisfaisant un ou plusieurs objectifs (coût, qualité, délai … etc.).

Dans un problème d'ordonnancement quatre notions fondamentales interviennent. Ce sont les tâches, les ressources, les contraintes et les objectifs [1.13].

- Tâche : elle est définit par un ensemble d'opération qui doivent être exécutées.
- Ressource : elle est un moyen matériel (machine) intervenant dans la réalisation d'une tâche. Il existe deux principaux types de ressources:
  - ✓ Les ressources renouvelables: elles sont disponibles en quantité constante tout au long de l'exécution des tâches. Les ressources renouvelables comme les machines, les processeurs, etc.
  - ✓ Les ressources consommables: une ressources est consommable si, après avoir été allouée à une tâche, elle n'est plus disponible pour les tâches restant à exécuter, c'est le cas de l'énergie par exemple.
- Contrainte: les contraintes représentent les limitations imposées par l'environnement ou les ressources.
- Objectif : ce sont les critères à optimiser.

La résolution d'un problème d'ordonnancement consiste à déterminer :

- ✓ Le placement des tâches dans l'espace, c'est-à-dire sur les processeurs ;
- ✓ Le placement des travaux dans le temps, c'est-à-dire les dates de début d'exécution de chaque tâche sur les ressources qui participent à sa réalisation ;

Les modes d'exécution :

Deux modes sont possibles :

- Avec préemption : l'exécution d'une tâche sur un processeur peut être interrompue puis reprise soit sur le même processeur ou sur un autre (migration de codes).
- Sans préemption : si une tâche a commencé, elle doit être menée jusqu'à son terme, sans interruption.

3.1.2. Temps de complétion maximal (makespan)

Makespan est l'un des critères d'optimisation les plus répandu. C'est le temps compris entre la date d'exécution de la première tâche d'une application et la date d'achèvement de la dernière tâche. Optimiser un ordonnancement suivant ce critère concourt à maximiser l'utilisation des ressources car cela revient à terminer l'exécution des applications dans le temps le plus court [1.10].

3.1.3. Heuristique

Le terme heuristique dérive du grec ancien « heuriskêin » et signifie ' trouver'.Il qualifie tout ce qui sert à la découverte et à l'exploitation. En informatique, une heuristique est une technique visant à accélérer la recherche d'une solution à un problème. Son but est d'aider à chercher dans la bonne direction et fait souvent appel à des connaissances "expertes".
Les heuristiques permettent généralement de faire un compromis entre la rapidité de la recherche et la qualité de la solution trouvée c.à.d.

- ✓ Trouver une solution optimale en un temps "pas trop long"
- ✓ Trouver rapidement une solution "pas trop mauvaise" [1.14].

3.1.4. Modélisation d'une application distribuée

Une application distribuée est souvent représentée par un ensemble de tâches qui ont des relations de précédences entre elles. Les relations de précédences donnent un ordre partiel d'exécution de tâches. Une tâche ne peut pas être exécutée tant que toutes les tâches

nécessaires à son exécution ne le sont pas. Lorsqu'elles le sont, la tâche devient prête à être exécutée. Cette dernière est qualifiée par l'adjectif « prête » [1.15]. Il existe différents modèles pour représenter une application sous forme d'un graphe comme les «Workflows» [1.13] (graphe de flots de données), « Task interaction graphs » (TIGs) [1.16] et le «Directed Acyclic Graph» (DAGs) (graphe orienté acyclique) que nous détaillons ci-après.

a. Modélisation d'une application par un DAG

Soit le graphe G= (N;A) où *N*= $\{n_1,\ldots..n_t\}$ est un ensemble de t nœuds ; *A*= $\{a_{i,j}\}$ est un ensemble d'arcs tel que $a_{i,j}$ est un arc partant depuis le nœud n_i vers le noeud n_j. S'il n'existe pas des cycles dans G alors il est appelé un DAG.

La représentation d'une application par un DAG se fait comme suit [1.15] :

- Les nœuds du DAG représentent les tâches qui composent l'application.
- L'existence d'un arc entre deux nœuds représente l'existence d'une contrainte de précédence entre les deux tâches associées aux nœuds.
- Chaque nœud du DAG est pondéré en fonction du temps d'exécution de chaque tâche.
- En plus des précédences, les tâches peuvent nécessiter des transferts de données. Pour prendre en compte les transferts de données dans la représentation, chaque arc est annoté d'une valeur qui correspond à la quantité de données.
- La tâche source dans le DAG est un nœud n'ayant aucun arc entrant, et la tâche puits est un nœud n'ayant aucun arc sortant.

Remarque : Il est clair qu'il ne peut pas y avoir de cycles dans les relations de précédences sans quoi certaines tâches ne pourraient être exécutées.

b. Recouvrement des communications

La parallélisation d'une application peut engendrer de nombreuses communications. Pour limiter l'impact de ces dernières, une méthode couramment utilisée est le recouvrement des communications par le calcul : durant la réalisation d'une communication, les machines exécutent des tâches. Le recouvrement des communications par les calculs implique que les durées de celles-ci seront connues à priori, lors de l'ordonnancement des tâches. Ces durées dépendront de la structure du réseau de communication, des caractéristiques des liens, des machines impliquées et des autres communications réalisées

en concurrence. Plusieurs modèles permettent une estimation de ces temps de communication plus ou moins réalistes.

Le modèle le plus simple existant est le modèle «délai», pour lequel le temps de communication est une fonction affine. Lorsqu'il n'y a pas de données à communiquer, le temps de transmission est la latence entre les machines impliquées. La pente de la fonction affine dépend du débit entre les machines. Pour que les communications suivent un tel modèle, la plate-forme doit comporter des liens de communications entre chaque machine. De plus, ce modèle est simple mais ne prend pas en compte plusieurs facteurs comme les coûts d'émission et de réception. Pour prendre en compte ces coûts, le modèle « LogP » a été développé par Culler et al. [1.17].

3.2. Ordonnancement

3.2.1. Les catégories d'ordonnancement

Les différentes catégories d'algorithmes d'ordonnancement sont [1.18]:

a. Placement statique ou dynamique

Dans le cas de placement statique, la décision de placement est prise hors ligne, c'est à dire avant l'exécution du programme. Les informations concernant l'état des ressources ainsi que de l'application soumise sont supposées disponibles au moment où l'ordonnancement est calculé. Ce type de placement est bien adapté aux applications déterministes dont le comportement est bien maitrisé d'une part, et d'autre part aux grilles pour lesquelles l'état des ressources peut être prédit de façon fiable, comme c'est le cas des grilles dont les ressources sont dédiées.

Le placement dynamique, quant à lui, repose sur le principe d'une allocation des tâches à la volée, lorsque l'application s'exécute. Il s'agit ainsi d'un placement en ligne. L'algorithme peut éventuellement s'adapter au cours de l'exécution de l'application à l'état des ressources, notamment à la charge des processeurs, en choisissant de migrer certaines tâches d'une ressource à une autre. Le placement dynamique est utile lorsqu'il est impossible, par exemple, de déterminer le temps d'exécution de l'application à exécuter.

b. Placement centralisé, décentralisé ou hiérarchique

Une stratégie d'ordonnancement centralisé concentre toutes les prises de décisions à un seul

endroit: un ordonnanceur unique est responsable de calculer un placement des applications soumises sur l'ensemble des ressources disponibles de la grille.

Cette approche est optimale, puisque l'ordonnanceur a une connaissance de l'état de toutes les ressources. Cependant, trois points négatifs se posent :

- le passage à l'échelle n'est pas facilité par une telle approche,
- ce système n'admet pas de tolérance aux fautes,
- le goulot d'étranglement ainsi crée peut engendrer une baisse importante des performances du système.

Le modèle décentralisé, ou distribue, confie la responsabilité de l'ordonnancement a différents ordonnanceurs, agissant de manière coopérative ou non. Enfin, la dernière stratégie d'ordonnancement repose sur un modèle hiérarchique, pour lequel les ordonnanceurs gèrent des entités de plus ou moins de hauts niveaux selon leur position dans la hiérarchie.
Ce dernier modèle constitue une combinaison de l'approche statique et de l'approche dynamique.

c. Modèle coopératif et modèle non coopératif

Dans le cas d'une stratégie d'ordonnancement dynamique ou hiérarchique, les différents ordonnanceurs peuvent collaborer entre eux pour déterminer un placement, ou bien ils peuvent travailler de manière totalement indépendante. Le modèle coopératif permet à chaque ordonnanceur de calculer une partie du placement global d'une application, en communiquant avec les autres ordonnanceurs dans le but d'évoluer vers un objectif commun.
Au contraire, les ordonnanceurs évoluant en mode non coopératif sont des entités autonomes qui prennent des décisions dans le but d'optimiser uniquement leurs propres objectifs, sans se soucier de la vue d'ensemble du système.

3.2.2. La dépendance des tâches d'une application

Les dépendances des tâches permettent de définir le moment où une tâche peut être lancée sur une machine. La dépendance à un impact crucial pour la conception des algorithmes d'ordonnancement,

a. Ordonnancement des tâches indépendantes

Dans ce type d'ordonnancement, toutes les informations sont connues à l'avance comme le temps d'exécution sur les différentes ressources, etc. Dans ce cas, les tâches s'exécutent en parallèle, indépendamment les unes des autres.

Dans la littérature, il existe plusieurs algorithmes d'ordonnancement de tâches indépendantes dans les paragraphes qui suivent, on introduit les plus connus[1.5] :

i. OLB (Opportunistic Load Balancing)

Dans cette ancienne méthode, la machine qui est inactif est sélectionnée sans tenir compte du temps d'exécution de la tâche. Si deux ou plusieurs machines sont au repos alors la machine est choisie arbitrairement. Dans cette méthode la période requise pour l'ordonnancement est moins et il garde presque toutes les machines occupées tout le temps possible.

ii. MET (Minimum Execution Time)

MET attribue chaque tâche à une ressource avec le meilleur temps d'exécution prévu pour cette tâche, peu importe si cette ressource est disponible ou non au temps demandé. La motivation derrière MET est de donner à chaque tâche sa meilleure machine. Ceci peut provoquer un déséquilibre de charge sévère dans les machines.

iii. MCT (Minimum Completion Time)

MCT attribue à chaque tâche, dans un ordre arbitraire, une ressource avec un temps d'achèvement minimum pour cette tâche. Cela provoque une certaine affectation des tâches à des machines qui n'ont pas le temps d'exécution minimum pour elles. L'intuition derrière MCT est de combiner les avantages de l'équilibrage de chargement opportuniste (OLB) et MET.

iv. Min-min

C'est une heuristique qui combine à la fois MET et MCT. Il sélectionne la tâche ayant le moins de temps exécution ensuite il assigne la tâche en utilisant MCT pour attribué la tâche à la ressource prévu pour la compléter au plus tôt.

v. Max-min

L'heuristique Max-min est similaire à l'heuristique Min-Min. Au lieu de choisir la plus petite tâche il sélectionne la tâche la plus grande en premier ensuite il applique MCT pour attribué la tâche à la ressource prévu pour la compléter au plus tôt.

b. Ordonnancement des tâches dépendantes

Les études sur l'ordonnancement d'applications composées de tâches dépendantes décrites par un DAG, se basent sur l'analyse des dépendances du graphe de tâches entier, afin de compléter les tâches interdépendantes au plus tôt. De nombreuses solutions d'ordonnancement proposées sont des heuristiques basées sur les poids des nœuds (temps d'exécution des tâches) et des arcs (temps de communication entre tâches) du DAG [1.17].

Dans ce qui suit, nous nous intéresserons aux algorithmes d'ordonnancement statiques des tâches dépendantes. Selon la méthode utilisée on distingue trois catégories d'algorithmes qui sont : les heuristiques de liste, d'agrégation de tâches (heuristiques clustérisés) et les heuristiques de duplication des tâches [1.5] et [1.15].

i. Les heuristiques de liste

Les heuristiques de liste sont toutes basées sur la succession des deux étapes suivantes :

1. Prendre une tâche parmi celles qui sont prêtes et/ou pas encore allouées à une machine qui se trouvent dans liste de tâches triées suivant une priorité. Une tâche est prête lorsque toutes les tâches parentes sont terminées et les données nécessaires aux calculs sont disponibles. Cette étape est aussi nommée phase de prioritisation car elle permet de faire un choix lorsqu'il existe plusieurs tâches disponibles. La priorité attribuée aux tâches permet de donner un ordre d'exécution.
2. Sélectionner une machine pour exécuter la tâche et allouer celle-ci à la machine.

Dans ce qui suit, nous présenterons quelques heuristiques comme EFT, Sufferage et HEFT. Cette dernière est abordée en détail puisque elle fera l'objet de notre étude dans le chapitre III.

- ***ETF** (Earliest Task First)*

L'objectif de cette heuristique est de minimiser les temps d'inactivités des processeurs à un instant donné, sans même se soucier de l'impact de ces choix sur la suite de l'exécution.

Ainsi lorsqu'un processeur devient inactif, l'algorithme sélectionne la prochaine tâche à exécuter. Pour chacune des tâches prêtes, l'algorithme calcule la date à laquelle la tâche considérée peut être exécutée le plus tôt possible. L'algorithme considère la date minimale avant laquelle aucune tâche ne peut être exécutée, Si un processeur devient inactif avant cette date, l'horloge virtuelle est avancée au prochain passage d'un processeur à l'état inactif. La liste de tâches prêtes est mise à jour et l'algorithme recommence le processus de sélection. [1.18]

- ***HEFT** (Heterogeneous-Earliest-Finish-Time)*

HEFT est parmi les algorithmes d'ordonnancements de liste les plus répandus. Il détermine un ordonnancement totalement statique d'un DAG sur un environnement hétérogène de manière à minimiser la durée totale d'exécution de l'application (makespan). [1.5]. Avant de présenter cet algorithme, nous avons besoin de quelques définitions :

Rang d'une tâche: c'est la hauteur (la hauteur d'une tâche est la longueur du chemin le plus long partant de celle-ci vers le puits) ou la distance maximale de la tâche par rapport au puits du DAG.

EFT (Earliest Finish Time): c'est une fonction qui cherche la ressource qui exécute la tâche le plus tôt possible.

Lors de l'ordonnancement, la liste de tâches prêtes est triée en fonction de rang des tâches. Plus une tâche à un rang important, plus elle est prioritaire. L'objectif de cet ordre est d'exécuter les tâches les plus éloignées de puits (nœud ayant aucun arc sortant) du DAG en premier pour éviter d'importants temps d'inactivité en fin d'exécution c.à.d. les ressources en attente d'une nouvelles tâches.

Le calcul de l'éloignement d'une tâche par rapport aux puits est effectué en considérant la longueur du chemin le plus long entre la tâche et le puits. La longueur du chemin prend en compte le coût de communications et les temps d'exécutions. Lors de ces calculs, il serait difficile de prendre en compte la congestion du réseau.

- *Sufferage*

L'objectif lors du placement des tâches sur les machines est de minimiser leur temps de démarrage dans l'ordre de la liste. Or, si l'ensemble des tâches de la liste est considéré, plusieurs d'entre elles obtiennent leur ordonnancement au plus tôt sur la même machine. Le problème est de savoir à quelle tâche sera allouée la machine.

Les heuristiques précédentes ont pour objectif d'optimiser l'exécution d'une tâche bien précise sans se soucier de l'impact de ce choix sur les temps d'exécution des autres. [1]
L'heuristique Sufferage est basée sur MCT. Le raisonnement derrière cette heuristique est qu'une tâche doit être affectée au processeur qui la rendra moins souffrir. La Sufferage est défini en tant que la différence entre la meilleure MCT et la deuxième meilleure MCT. Lors d'ordonnancement, l'algorithme donne la plus grande priorité aux tâches qui ont la plus grande valeur Sufferage [1.8].

ii. Les heuristiques d'agrégation de tâches

Les algorithmes d'agrégation de tâches («task clustering») ont été développés dans l'idée que le nombre de machines sera suffisant pour exploiter la totalité du parallélisme des applications. Ainsi dans le problème d'ordonnancement, l'algorithme peut supposer que le nombre de machines est infini. Dans ce cas, le placement d'une tâche par machine est une solution envisageable. Avec un tel ordonnancement, le temps d'exécution est égal au chemin critique de l'application en accumulant le temps d'exécution des tâches et les temps de communications. Pour minimiser le temps d'exécution global, l'algorithme peut influencer le placement des tâches. Il peut ainsi regrouper l'exécution de certaines tâches sur la même machine pour éviter les coûts de communications entre deux tâches. L'objectif est de limiter les communications qui impactent le temps d'exécution global sur une infinité de machines. [1.15]

- *EZ (Edge Zeroing)*

Pour cet algorithme, les arêtes du DAG sont triées dans l'ordre décroissant des coûts de communications. Itérativement dans l'ordre décroissant, l'algorithme cherche pour chaque arête à connaître l'impact de l'agrégation de la tâche qui produit les données et de celle qui les consomme. Dans un premier temps, nous supposons que chacune des tâches est seule dans l'agrégat les contenant. Si les tâches sont exécutées par la même machine, la communication entre les tâches est considérée comme nulle. Pour évaluer l'impact sur le temps d'exécution, il suffit de comparer la longueur des chemins critiques (un chemin critique est modélisé par la tâche avec le temps d'exécution le plus important) des deux DAG : celui avec l'agrégation et celui sans. Si la longueur du chemin critique est inférieure ou égale à celle du DAG sans agrégation, elle est conservée.

Nous avons expliqué le principe pour l'agrégation pour deux tâches, mais il peut y en avoir plusieurs dans chaque agrégat. Dans ce cas, les agrégats sont fusionnés en un seul. Lors de cette fusion, certaines des tâches appartenant aux deux agrégats peuvent être indépendantes (il n'existe pas de chemin liant les deux tâches). L'ordre d'exécution de ces tâches par la machine qui exécutera l'agrégat fusionné doit être défini. Ce choix peut avoir un impact sur le temps d'exécution.

Pour choisir cet ordre, on a proposé une heuristique basée sur la hauteur de la tâche (b_level). La hauteur des tâches est calculée avant la fusion des deux agrégats.
Celle-ci définit l'ordre d'exécution des tâches indépendantes : les tâches les plus hautes sont exécutées en premier. Une arête fictive (sans communication) est ajoutée entre les tâches indépendantes pour définir un ordre.

En fonction de cet ordre, le temps d'exécution du DAG en parallèle est calculé en considérant que l'agrégat résultant de la fusion est exécuté par une machine. De manière équivalente, ce temps est mesuré avec la longueur du chemin critique en fournissant un poids nul aux arêtes incluses dans un agrégat.

Si le temps parallèle avec cette agrégation est inférieur ou égal au temps d'exécution sans effectuer la fusion, l'agrégation est conservée pour l'étape suivante. [1.15]

- ***CTHP*** *(Clustering Tasks onto Heterogeneous Processors)*

Cet algorithme est constitué de deux étapes. Initialement la stratégie suppose un environnement virtuel homogène tout en créant des groupes de tâches en utilisant la structure de l'algorithme de clusterisation. Dans une deuxième étape, un sous-ensemble des clusters déjà générés sont plaqués à l'environnement hétérogène, compte tenu du coût d'exécution des tâches sur les processeurs et des caractéristiques de transmission sur les liens de communication [1.13].

ii. Les heuristiques de duplication des tâches

Les heuristiques à duplication de tâches consistent à allouer de manière redondante certaine tâches « importantes » dont d'autres dépendent. Le but recherché est donc de réduire le temps avant que les tâches en attente puissent commencer ce qui peut éventuellement améliorer le temps d'exécution global de l'application [1.13].

- ***TDS (Task Duplication-based Scheduling)***

TDS ordonnance un DAG sur des processeurs hétérogènes interconnectés par des liens homogènes de communication. L'heuristique est composée de trois étapes. Initialement, pour chaque tâche du graphe les temps de début et de fin au plus tôt sont calculés en ordre de manière à identifier le prédécesseur qui devrait être ordonnancé avec la tâche et le processeur qui réduisent au maximum son temps de terminaison. La prochaine étape consiste à ordonnancer les tâches (dans l'ordre de leur niveau), basée sur le prédécesseur et le processeur favori, et sur les temps de début et de fin au plus tard de la tâche sélectionnée. Dans la dernière étape, une procédure de duplication tente de reproduire le prédécesseur favori de la tâche sélectionnée sur son processeur si le makespan diminue [1.13].

Remarque: Les algorithmes d'ordonnancement des tâches indépendantes, tel que Min-min et Max-min, ont été adapté pour les applications aux tâches avec précédences. Aussi d'autres algorithmes, qui seront discutés dans le chapitre suivant, basés sur des métaheuristiques tel que les algorithmes génétique et les colonies de fourmis ont été utilisés pour résoudre le problème d'ordonnancement.

4. Conclusion

Les grilles informatiques sont des technologies qui ne sont pas récentes, mais elles suscitent de plus en plus l'engouement à travers le monde. En effet, elles représentent une fabuleuse alternative au calcul intensif classique.

Dans ce chapitre, nous avons présenté les notions en rapport avec les grilles et nous nous somme intéressé au problème de l'ordonnancement des tâche sous ce types d'architecture en exposant quelques algorithmes d'ordonnancement existant dans la littérature.

Table des matières

1. Introduction

L'intelligence artificielle (IA) a été conçue pour apporter des réponses à des problèmes complexes, qui peuvent avoir un grand nombre de solutions possibles. Cela est dû à la variété des méthodes utilisées pour la résolution de ce problème. Le problème qui se pose est la manière de trouver le meilleur choix parmi toutes les solutions.

Pour résoudre ce problème, les chercheurs en IA ont proposé des méthodes ingénieuses, appelées les heuristiques, qui permettent de trier parmi les solutions possibles, afin d'accélérer le processus de sélection de la meilleure solution.

Les techniques d'intelligence artificielle (réseaux de neurones, les algorithmes génétiques, l'optimisation par essaim de particules…etc.) ont été développées à partir de la compréhension de phénomènes biologiques et naturels. Ils suivent souvent des lois stochastiques, ce qui les rend plus proches de la réalité, car la plus part des phénomènes physiques suivent le hasard.

Dans ce chapitre, nous présenterons un état de l'art de quelques techniques d'intelligence artificielle existantes dans la littérature. Une attention particulière est accordée à la méthode d'optimisation par essaim de particules que nous allons utilisée, par la suite, dans notre problème qui est l'ordonnancement des tâches dans une grille de calcul.

2. Etat de l'art des métaheuristiques d'optimisation

2.1. Problème d'optimisation

Un problème d'optimisation en général est défini par un espace de recherche *S* et une fonction objectif *f* . Le but est de trouvé une solution $s^* \in S$ de meilleure qualité *f*(s). Suivant le problème posé, on cherche soit le minimum ou le maximum de la fonction *f*. L'équation suivante résume la définition : $s^* = \min (f(s) \setminus s \in S)$.

Il existe de nombreuses méthodes déterministes ou exactes qui permettent de résoudre certains types de problèmes d'optimisation en un temps fini. D'autres problèmes restent cependant trop complexes à résoudre par ces méthodes. Dans ce cas, le problème d'optimisation est dit « difficile », car aucune méthode déterministe ne peut le résoudre en un temps « raisonnable » [2.1].

2.2. Définition de Métaheuristique

Les métaheuristiques sont une famille d'algorithmes stochastiques destinés à la résolution des problèmes d'optimisation. Leurs particularités résident dans le fait que celle-ci sont adaptables à un grand nombre de problèmes sans changement majeur dans leurs algorithmes, d'où le qualificatif '**méta**'. Leur capacité à résoudre un problème à partir d'un nombre minimal d'informations est contrebalancée par le fait qu'elles n'offre aucune garantie quant à l'optimalité de la meilleure solution trouvée. Seule une approximation de l'optimum global est donnée. Cependant ce constat n'est pas forcément un désavantage, étant donné qu'on préférera toujours une approximation de l'optimum global trouvé rapidement qu'une valeur exacte trouvé dans un temps rédhibitoire.

Les métaheuristiques sont des méthodes qui ont, en général, un comportement itératif c.à.d. le même schéma est reproduit un certain nombre de fois au cours de l'optimisation. Ces méthodes tirent leurs efficacités du fait qu'elles sont moins sujettes à être piégées dans des optima locaux, car elles acceptent, au cours de traitement, des dégradations de la fonction objectif et la recherche est souvent menée par une population de points et non par un point unique.[2.1]

Les métaheuristiques se caractérisant par leur capacité à résoudre des problèmes très divers, elles se prêtent naturellement à des extensions. Pour illustrer celles-ci, nous pouvons citer [4] :

- Les métaheuristiques pour l'optimisation **multi-objectif** où il faut optimiser plusieurs objectifs contradictoires. Le but ne consiste pas ici à trouver un optimum global, mais à trouver un ensemble d'optima, qui forment une surface de compromis pour les différents objectifs du problème ;
- Les métaheuristiques pour l'optimisation **multimodale** où l'on ne cherche plus l'optimum global, mais l'ensemble des meilleurs optima globaux et/ou locaux ;
- Les métaheuristiques pour l'optimisation de problèmes bruités : où il existe une incertitude sur le calcul de la fonction objectif, dont il faut tenir compte dans la recherche de l'optimum ;
- Les métaheuristiques pour l'optimisation **dynamique** où la fonction objectif varie dans le temps, ce qui nécessite d'approcher l'optimum à chaque pas de temps ;

- Les métaheuristiques **hybrides** qui consistent à combiner différentes métaheuristiques, afin de tirer profit des avantages respectifs ;
- Les métaheuristiques **parallèles** où l'on cherche à accélérer le calcul, en répartissant la charge de calcul sur des unités fonctionnant de concert. Le problème revient alors à adapter les métaheuristiques pour qu'elles soient distribuées.

2.3. Quelques exemples de métaheuristiques

Nous présentons, dans cette section, quelques métaheuristiques qui sont largement utilisées [2.2]:

2.3.1. Les algorithmes génétiques

Les algorithmes génétiques sont des algorithmes d'optimisation s'appuyant sur des techniques dérivées de la génétique et de l'évolution naturelle : croisements, mutations, sélections, etc. Pour les utiliser, on doit disposer des cinq éléments suivants :

1. Un principe de codage. On associe à chacun des points de l'espace d'état une structure de données. La qualité du codage conditionne le succès des algorithmes génétiques.
2. Un mécanisme de génération de la population initiale qui doit être capable de produire une population d'individus non homogène qui servira de base pour les générations futures.
3. Une fonction à optimiser. Celle-ci retourne une valeur appelée fitness ou fonction d'évaluation de l'individu.
4. Des opérateurs (croisement, mutation) permettant de diversifier la population au cours des générations et d'explorer l'espace d'état.
5. Des paramètres de dimensionnement : taille de la population, nombre total de générations ou critère d'arrêt, probabilités d'application des opérateurs de croisement et de mutation.

Le principe général du fonctionnement d'un algorithme génétique est représenté sur la figure

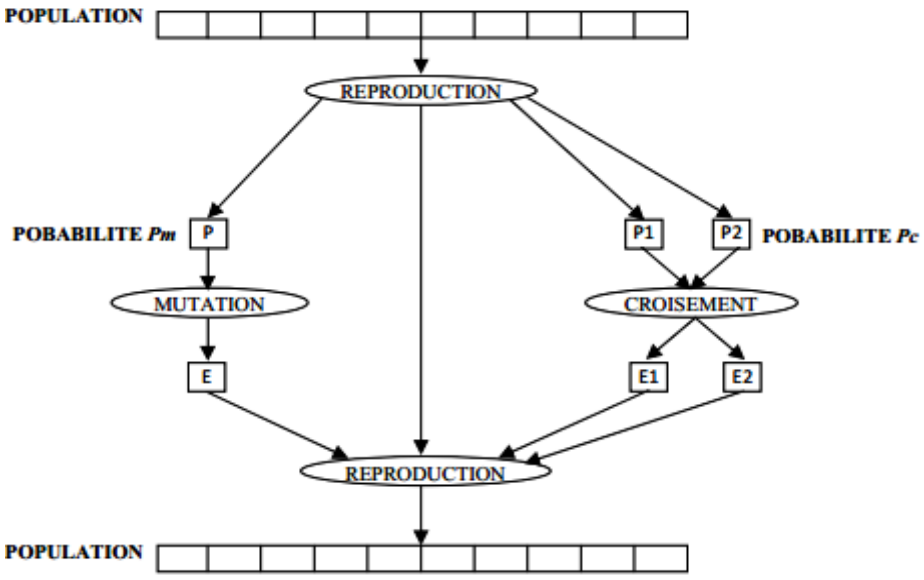


Figure 2.1 : Principe général des AGs.

2.3.2. Les réseaux de neurones

Un réseau de neurones artificiels est à l'origine une tentative de modélisation mathématique du cerveau humain. Ainsi, il représente un modèle de calcul dont la conception est très schématiquement inspirée du fonctionnement des neurones biologiques.

Ils sont généralement optimisés par des méthodes d'apprentissage de type probabiliste. Ils sont placés d'une part dans la famille des applications statistiques, qu'ils enrichissent avec un ensemble de paradigmes permettant de créer des classifications rapides, et d'autre part dans la famille des méthodes de l'intelligence artificielle auxquelles ils fournissent un mécanisme perceptif indépendant des idées propres de l'implémenteur, et fournissant des informations d'entrée au raisonnement logique formel.
Le neurone calcule la somme pondérée de ses entrées puis cette valeur passe à travers la fonction d'activation pour produire sa sortie.

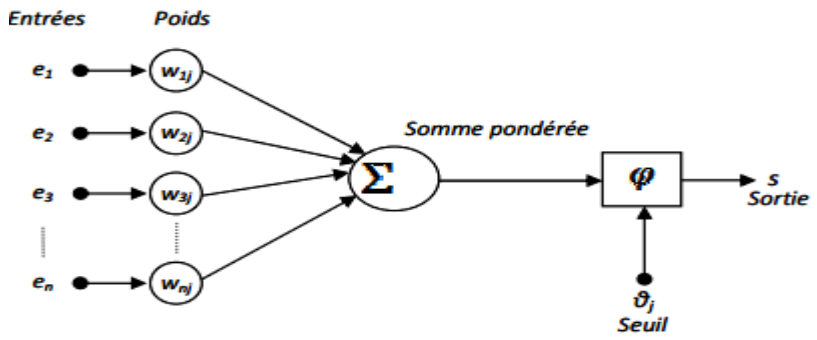


Figure 2.2 : Structure d'un neurone artificiel.

2.3.3. Les algorithmes de colonies de fourmis

Les algorithmes de colonies de fourmis sont nés d'une constatation simple : les insectes sociaux et en particulier les fourmis résolvent naturellement des problèmes complexes. Un t el comportement est possible car les fourmis communiquent entre elles de manière indirecte par le dépôt de substances chimiques, appelées phéromones, sur le sol. Ce type de communication indirecte est appelée **stigmergie**. La principale illustration de ce constat est donnée par la figure suivante.

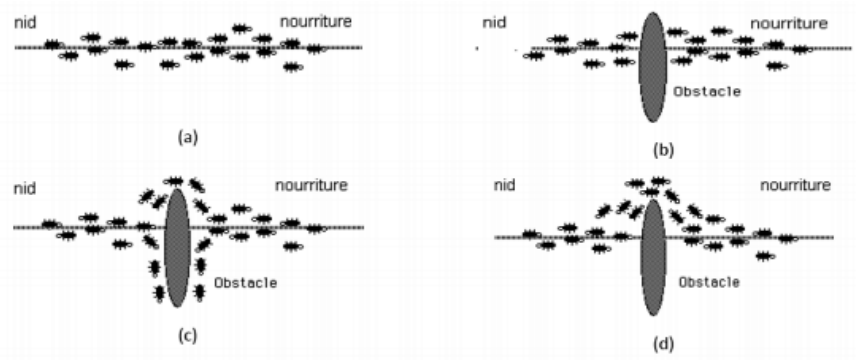


Figure 2.3: Détermination du plus court chemin par une colonie de fourmis.
(a) Situation initiale, (b) Introduction d'un obstacle, (c) Recherche du chemin optimal, (d) Prédominance du chemin optimal.

On voit sur cette figure que si un obstacle est introduit sur le chemin des fourmis, celle-ci vont, après une phase de recherche, avoir tendance à toutes emprunter le plus court chemin entre le nid et l'obstacle. Plus le taux de phéromone à un endroit donné est important, plus une fourmi va avoir tendance à être attirée par cette zone. Les fourmis qui sont arrivées le plus rapidement au nid en passant par la source de nourriture sont celles qui ont emprunté la branche la plus courte du trajet. Il en découle donc que la quantité de phéromone sur ce trajet est plus importante que sur le trajet plus long. De ce fait, le plus court chemin a une probabilité plus grande d'être emprunté par les fourmis que les autres chemins et sera donc, à terme, emprunté par toutes les fourmis.

Nous détaillerons dans la section suivante une métaheuristique (optimisation par essaim particulaire) que nous allons utiliser par la suite.

2.4. Optimisation par Essaim Particulaire

2.4.1. Principe général

L'optimisation par essaim particulaire (OEP), ou Particle Swarm Optimization (PSO) en anglais, est un algorithme évolutionnaire qui utilise une population de solutions candidates pour développer une solution optimale au problème. Cet algorithme a été proposé par Russel Eberhart (ingénieur en électricité) et James Kennedy (socio-psychologue) en 1995. Il s'inspire à l'origine du monde du vivant, plus précisément du comportement social des animaux évoluant en essaim, tels que les bancs de poissons et les vols groupés d'oiseaux.
L'essaim de particules correspond à une population d'agents simples, appelés particules. Chaque particule est considérée comme une solution du problème, où elle possède une position (le vecteur solution) et une vitesse. De plus, chaque particule possède une mémoire lui permettant de se souvenir de sa meilleure performance (en position et en valeur) et de la meilleure performance atteinte par les particules « voisines » (informatrices) : chaque particule dispose en effet d'un groupe d'informatrices, historiquement appelé son voisinage. Un essaim de particules, qui sont des solutions potentielles au problème d'optimisation, « survole » l'espace de recherche, à la recherche de l'optimum global [2.3].

L'algorithme PSO peut être décrit en se plaçant du point de vue d'une Particule. Au départ de l'algorithme, un essaim est réparti au hasard dans l'espace de recherche, chaque particule ayant également une vitesse aléatoire. Ensuite, à chaque pas de temps [2.2]:

- Chaque particule est capable d'évaluer la qualité de sa position et de garder en mémoire sa meilleure performance, c'est à dire la meilleure position qu'elle a atteinte jusqu'ici (qui peut en fait être parfois la position courante) et sa qualité (la valeur en cette Position de la fonction à optimiser).
- Chaque particule est capable d'interroger un certain nombre de ses congénères (ses informatrices, dont elle même) et d'obtenir de chacune d'entre elles sa propre meilleure performance (et la qualité afférente).
- à chaque pas de temps, chaque particule choisit la meilleure des meilleures performances dont elle a connaissance, modifie sa vitesse en fonction de cette information et de ses propres données et se déplace en conséquence.

Pour réaliser son prochain déplacement, une particule est influencée par les trois composantes suivantes [2.3]:

1. Une composante d'inertie : la particule tend à suivre sa direction courante de déplacement.
2. Une composante cognitive : la particule tend à se diriger vers le meilleur site par lequel elle est déjà passée ;
3. Une composante sociale : la particule tend à se fier à l'expérience de ses congénères et, ainsi, à se diriger vers le meilleur site déjà atteint par ses voisins.

La stratégie de déplacement d'une particule est illustrée dans la figure suivante :

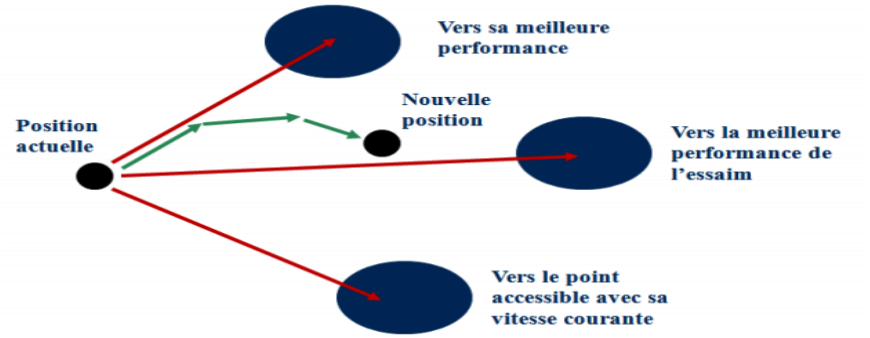


Figure 2.4 : Déplacement d'une particule.

La figure suivante présente un schéma de principe résumant les explications ci-dessus.

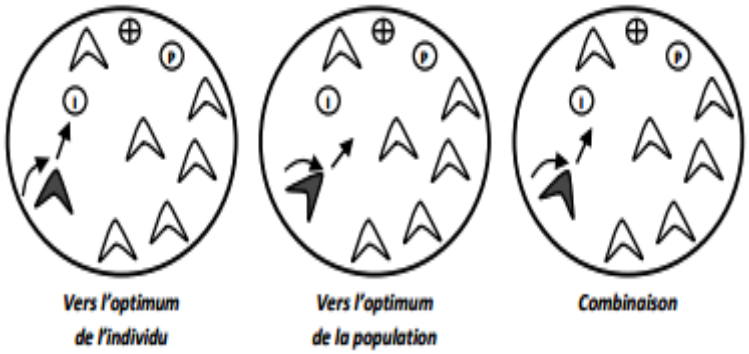


Figure 2.5 : Règles simples d'application locale utilisées dans un essaim de particules [2.2].

Où :

- I : est l'optimum de l'individu.
- P : est l'optimum de la population.
- + : est l'optimum absolu.

2.4.2. Formalisation

Dans un espace de recherche de dimension D, la particule i de l'essaim est modélisée par :

- un vecteur position $\overrightarrow{X}_i = (x_{i1}, x_{i2}, \ldots, x_{iD})$
- un vecteur vitesse $\overrightarrow{V}_i = (v_{i1}, v_{i2}, \ldots, v_{iD})$.

La qualité de sa position est déterminée par la valeur de la fonction objectif en ce point.
Chaque particule garde en mémoire :

- la meilleure position par laquelle elle est déjà passée, notée $\overrightarrow{Pbest}_i = (pbest_{i1}, pbest_{i2}, \ldots, pbest_{iD})$.
- La meilleure position atteinte par les particules de l'essaim notée $\overrightarrow{Gbest} = (gbest_1, gbest_2, \ldots, gbest_D)$.[1]

Remarque : Le terme de « vitesse » est abusif, car les vecteurs $\overrightarrow{V}_i$ ne sont pas homogènes à une vitesse. Il serait plus approprié de parler de « direction de déplacement ». Cependant, pour respecter l'analogie avec le monde animal, les auteurs ont préféré utiliser le terme de « vitesse» [4].
Au départ de l'algorithme, les particules de l'essaim sont initialisées de manière aléatoire dans l'espace de recherche du problème. Ensuite, à chaque itération, chaque particule se déplace, en combinant linéairement les trois composantes citées ci-dessus. En effet, à l'itération (t + 1), le vecteur vitesse et le vecteur position sont calculés à partir de l'équation (1) et de l'équation (2), respectivement.

$$v_{i,j}^{t+1} = w v_{i,j}^{t} + c_1 r_{1\,i,j}[pbest_{i,j}^{t} - x_{i,j}^{t}] + c_2 r_{2\,i,j}[gbest_j^{t} - x_{i,j}^{t}], \quad j \in \{1,2,\ldots D\} \qquad (1).$$

$$x_{i,j}^{t+1} = x_{i,j}^{t} + v_{i,j}^{t+1}, \quad j \in \{1,2,\ldots D\} \qquad (2).$$

où w est une constante, appelée *coefficient d'inertie* ; $c_1$et $c_2$sont deux constantes, appelées, *coefficients d'accélération* ; r_1 et $r_2$sont deux nombres aléatoires tirés uniformément dans$[0, 1]$, à chaque itération t et pour chaque dimension j.
Les trois composantes mentionnées ci-dessus (i.e. d'inertie, cognitive et sociale) sont représentées dans l'équation (1) par les termes suivants :

1. $w v_{i,j}^{t}$ correspond à la composante d'inertie du déplacement, où le paramètre w contrôle l'influence de la direction de déplacement sur le déplacement futur ;
2. $c_1 r_{1\,i,j}[pbest_{i,j}^{t} - x_{i,j}^{t}]$ correspond à la composante cognitive du déplacement, où le paramètre c_1 contrôle le comportement cognitif de la particule ;
3. $c_2 r_{2\,i,j}[gbest_j^{t} - x_{i,j}^{t}]$ correspond à la composante sociale du déplacement, où le paramètre c_2 contrôle l'aptitude sociale de la particule.

Une fois le déplacement des particules effectué, les nouvelles positions sont évaluées et les deux vecteurs $\overrightarrow{Pbest}_i$ et $\overrightarrow{Gbest}$ sont mis à jour, à l'itération $t + 1$, suivant les deux équations

[1] Nous nous référons à la version globale (premier algorithme proposé par les auteurs) de PSO c.à.d. topologie entièrement connectée, où toutes les particules de l'essaim sont considérées comme voisines de la particule i, d'où la notation Gbest (global best).

(3) (dans le cas d'une minimisation) et (4) (dans une version globale de PSO), respectivement, où N est le nombre de particules de l'essaim.

$$\overrightarrow{Pbest}_i^{t+1} = \begin{cases} \overrightarrow{Pbest}_i^t, \ si\ f(\overrightarrow{X}_i^{t+1}) \geq \overrightarrow{Pbest}_i^t \\ \overrightarrow{X}_i^{t+1}, \ sinon \end{cases} \quad (3).$$

$$\overrightarrow{Gbest} = \arg\ min_{\overrightarrow{Pbest}\,i}\ f(\overrightarrow{Pbest}_i^{t+1}),\ 1 \leq i \leq N. \quad (4).$$

1. **Initialiser** aléatoirement N particules : position $\overrightarrow{X}_i$ et vitesse $\overrightarrow{V}_i$.
2. **Evaluer** les positions des particules en utilisant la fonction objectif.
3. **Pour** chaque particule i, $\overrightarrow{Pbest}_i = \overrightarrow{X}_i$
4. **Calculer** Gbest selon (4)
5. **tant que** le critère d'arrêt n'est pas satisfait **faire**
6. **Déplacer** les particules selon (1) et (2)
7. **Evaluer** les positions des particules
8. **Mettre à jour** (3) et (4)
9. **fin**

Figure 2.6 : L'algorithme de PSO.

La figure 2.7 donne une illustration d'un organigramme de l'algorithme de PSO.

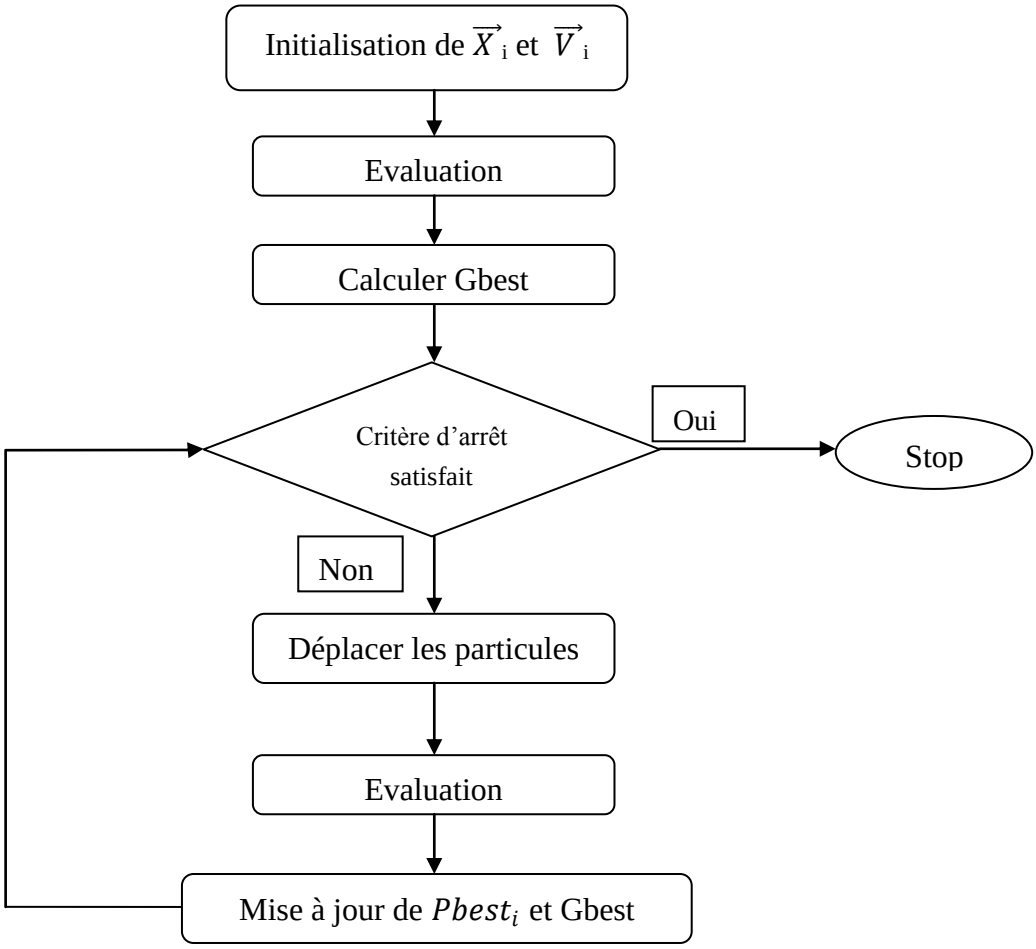


Figure 2.7 : Organigramme de PSO.

2.4.3. Améliorations de PSO

Depuis son apparition, plusieurs améliorations et des nouvelles notions ont été apportées à la version de PSO précédemment décrite appelée la version globale [2.3].

a. Confinement des particules

Pour éviter que le système n'« explose » en cas d'amplification trop grande d'oscillations (il est possible que le déplacement d'une particule soit trop rapide et la conduise à sortir de l'espace de recherche), un nouveau paramètre $\boldsymbol{V_{max}}$ est introduit. Il permet de limiter la vitesse sur chaque dimension et ainsi de contrôler l'explosion du système [2.3]. Notons que cela ne restreint pas les valeurs des positions à l'intervalle [V_{min} , V_{max}], mais limite seulement la distance maximale qu'une particule va parcourir au cours d'une itération. Cette méthode permet de contrôler la divergence de l'algorithme et de réaliser ainsi un compromis efficace entre intensification et diversification.

De plus, une stratégie de confinement des particules peut être introduite. Une telle stratégie permet de ramener une particule sortie de l'espace de recherche à l'intérieur de celui-ci. Dans ce cadre, plusieurs méthodes peuvent être employées comme :

- La particule est laissée à l'extérieur de l'espace de recherche, mais on n'évalue pas sa fonction objectif. Ainsi, elle ne pourra pas attirer les autres particules en dehors de l'espace de recherche ;

- La particule est arrêtée à la frontière et les composantes associées à sa vitesse sont annulées ;
- La particule rebondit sur la frontière. La particule est stoppée sur la frontière, mais les composantes correspondantes de la vitesse sont multipliées par un coefficient tiré aléatoirement dans l'intervalle [-1,0] pour revenir en arrière.

b. Coefficient d'inertie

Le coefficient d'inertie w contrôle l'influence de la direction de la particule sur le déplacement futur. Le but de l'introduction de ce paramètre est de réaliser un équilibre entre la recherche locale (exploitation) et la recherche globale (exploration). L'intensité de l'exploration de l'espace de recherche dépend de la valeur du poids d'inertie, une grande valeur de w facilitant une exploration globale, alors qu'une petite valeur facilite l'exploration locale. Du fait de son influence sur les performances de l'algorithme PSO, le poids d'inertie a suscité un grand intérêt de la part de la communauté des chercheurs. Dans la littérature, un coefficient d'inertie dynamique qui varie au cours du temps a été proposé. Il commence par une valeur proche de 0,9 et descend linéairement pour arriver à 0,4. Cette stratégie a beaucoup amélioré les performances de PSO pour plusieurs problèmes d'optimisation. Le coefficient d'inertie w varie linéairement avec le temps selon la formule suivante : $w = w_{min} + (w_{max} - w_{min}).(iter/max_{iter})$ (5)

Où : iter est l'itération courante.

max_{iter} est le nombre maximal d'itérations.

w_{max} et w_{min} désignent respectivement les valeurs maximum et minimum du coefficient w (généralement, w_{max} , w_{min} $\in[0, 1]$).

c. Topologie de voisinage

Comme nous l'avons vu, l'algorithme PSO est inspiré du comportement collectif des essaims lors des études et des simulations informatiques de groupes d'animaux (vols d'oiseaux ou bancs de poissons). Ces études mettent en évidence la capacité d'un individu de rester à une distance optimale par rapport aux autres dans le même groupe et à suivre un mouvement global affecté par les mouvements locaux de ses voisins. À partir de ces simulations, les auteurs ont modélisé le comportement des individus par les équations (1) et (2). Dans la pratique, en utilisant l'équation (1), un réseau doit être défini afin d'établir des connexions entre les particules et leur permettre d'échanger des informations entre elles. Ce réseau de communication entre les particules est appelé topologie de voisinage. Cette topologie contribue à définir un groupe d'informatrices pour chaque particule, c'est ce que l'on appelle le voisinage d'une particule. Le voisinage d'une particule peut donc être défini comme le sous-ensemble de particules de l'essaim avec lesquelles celle-ci a une communication directe, i.e. chaque particule peut interroger les particules dans son voisinage (ses informatrices) qui, à leur tour, lui transmettent leurs informations.

A l'origine, dans la version de PSO résumée dans l'Algorithme de PSO déjà décrit dans la figure 2.6, les auteurs ont défini une topologie entièrement connectée (i.e. chaque particule est reliée à toutes les autres). Cette version de PSO est appelée version globale (Gbest), car la particule est informée par la totalité des autres, et l'information effectivement utilisée est incarnée par le terme Gbest de la troisième composante de l'équation (1). Cette version a l'inconvénient majeur de ne pas donner lieu à une exploration suffisante de l'espace de recherche, ce qui peut conduire à une stagnation dans un optimum local et donc à une

convergence prématurée. De nombreuses variantes de la version originale, dites versions locales (Lbest), ont été proposées dans la littérature de PSO, afin d'améliorer sa convergence. Dans les versions locales, le terme G_{best} est remplacé par le terme$L_{best\ i}$, où, pour chaque particule i, on définit un ensemble de voisinage (i.e. l'information qui doit être partagée est la meilleure solution trouvée dans le voisinage de chaque particule).

i. Voisinage géographique et social

La notion de voisinage peut avoir deux types principaux, géographique et social, qui donnent aux particules une capacité de perception de leur environnement proche, selon le type utilisé (i.e. l'environnement proche peut être défini par un réseau social ou un réseau géographique).

Le voisinage géographique d'une particule i se compose des particules proches. Pour considérer cette proximité, la distance entre la particule i et les autres particules de l'essaim est utilisée, sachant que le nombre de voisins doive être défini à l'avance.

Le voisinage social d'une particule est le sous-ensemble des voisins virtuels défini au début de l'algorithme PSO ; il n'est pas nécessairement fixe et peut être adaptatif. Il peut être représenté sous la forme d'un graphe, où les particules sont des sommets reliés à leurs voisines par les arêtes du graphe. Le but d'utiliser ce type de voisinage est d'améliorer la rapidité de convergence de l'algorithme. Cela signifie qu'il est nécessaire de contrôler la propagation de l'information dans le graphe. Pour ce faire, il s'agit donc de trouver quels sont les bons paramètres (par exemple, la taille du voisinage, la forme du réseau social, la ou les sources d'influence) qui structurent le graphe et qui sont responsables de la vitesse de propagation du flux d'information. Savoir choisir la topologie est donc fondamental et demande une adaptation au problème traité. La figure suivante montre la différence entre un voisinage géographique et un voisinage social.

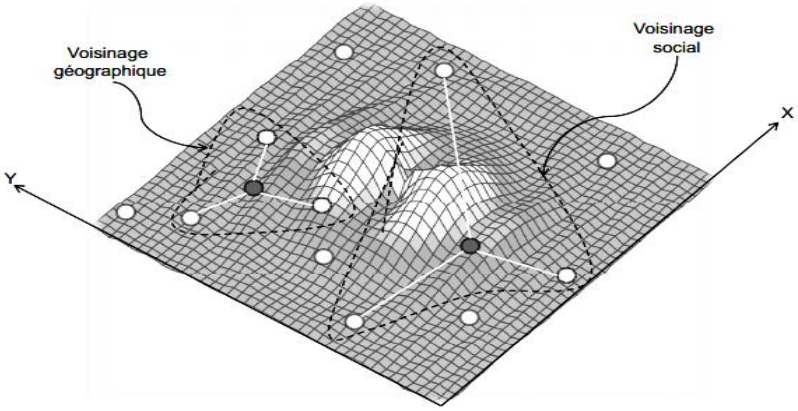


Figure 2.8 : Voisinage géographique et voisinage social. [2.3]

ii. Les modèles de topologies

PSO a connu deux versions originales, juste après son développement en 1995, qui sont la version globale (Gbest) et la version locale classique en anneau (Lbest classique).

En plus de ces deux modèles originaux, un grand nombre de topologies ont été proposées et beaucoup d'entre elles ont laissé un impact positif sur l'algorithme PSO en termes de performances. Nous distinguons les topologies statiques des topologies dynamiques, avant de

décrire, ci-dessous, une sélection de ces topologies, qui ont été citées plusieurs fois dans la littérature et ont eu un certain succès.

1. Topologies statiques

A l'origine, l'algorithme PSO n'utilisait pas de topologies dynamiques. Celles utilisées étaient structurées d'une manière régulière ou sur des formes géométriques. Ces topologies sont donc appelées graphes statiques, car elles sont définies une fois pour toutes. Les versions statiques les plus connues (celles qui sont principalement utilisées) sont la version globale Gbest (illustré dans la figure 2.9 (a).) et la version locale classique sous forme d'un anneau (Lbest classique) (illustré dans la figure 2.9 (b).), où chaque particule n'a que deux particules voisines. Dans le modèle Lbest classique, l'essaim converge plus lentement que dans Gbest, mais il a plus de chances de localiser l'optimum global. En général, l'utilisation du modèle Lbest permet de limiter les risques d'une convergence prématurée de l'algorithme. Ce qui a un effet positif sur les performances de l'algorithme, surtout pour les problèmes multimodaux. Plusieurs topologies statiques ont été proposées, nous présentons ci-dessous quelques-unes d' entre elles :

- **Wheel ou étoile**: Cette topologie a la forme d'une roue, où la particule du centre de l'essaim, nommée focale (principale), est responsable du flux des informations. Dans cette topologie, toutes les particules sont isolées les unes des autres et elles sont liées uniquement à la particule focale. Les informations doivent être communiquées à travers cette particule, qui va ensuite utiliser ces informations pour adapter sa trajectoire. Si le résultat des ajustements montre une amélioration des performances de la particule principale, alors cette amélioration sera communiquée au reste de la population. Donc la particule principale sert de filtre, qui ralentit la vitesse de transmission de l'information sur la meilleure solution trouvée. La figure 8 (c) illustre une représentation graphique du modèle Wheel.
- **Von Neumann**: elle est communément adoptée sur les deux modèles Gbest et Lbest classique. Dans Von Neumann, la topologie prend la forme d'une grille (réseau rectangulaire), où chaque particule est reliée à ses quatre particules voisines (à gauche, à droite, au dessus et en dessous). Comme le modèle Lbest classique, cette topologie ralentit la vitesse de convergence de l'algorithme en raison de sa structure particulière, qui ralentit la propagation de l'information à toutes les particules de l'essaim. Une représentation graphique du modèle de Von Neumann est illustrée dans la figure 2.9 (d).
- **Four-clusters**: La topologie Four-clusters utilise quatre groupes de particules reliés entre eux par plusieurs passerelles. Du point de vue sociologique, cette topologie ressemble à quatre communautés isolées. Dans chaque communauté, quelques particules peuvent communiquer avec une particule d'une autre communauté. Les particules sont divisées en quatre groupes, chaque groupe est composé de cinq particules, qui échangent l'information avec les trois autres groupes via les passerelles. Chaque passerelle n'est liée qu'à un seul groupe. Cette topologie est caractérisée par le nombre d'arêtes qui relient indirectement deux particules. Le diamètre du graphe est de trois, ce qui signifie que l'information peut être transmise d'une particule à une autre en empruntant au maximum trois arêtes. Grâce à ce diamètre, chaque particule

peut obtenir une vision sur l'information globale, en effectuant au maximum trois itérations de l'algorithme. La figure 8 (e) montre la représentation graphique de ce modèle.

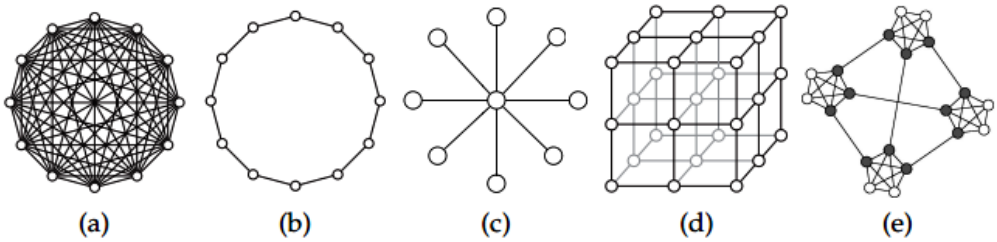


Figure 2.9 : Topologies statiques.

2. Topologies dynamiques

En général, les topologies statiques ne résolvent pas le problème de la convergence prématurée de PSO même elles améliorent ses performances. Alors des topologies dynamiques ont été mises en place. Ces topologies pouvaient avoir de meilleures performances que les topologies statiques, et ce, au moyen d'un changement de leurs structures, d'une itération à l'autre. Ce changement permet aux particules de changer leurs trajectoires de déplacement, afin de pouvoir s'échapper des optima locaux. Nous citons ci-dessous quelques topologies dynamiques :

- **Fitness**: les particules sont disposées dans un anneau régulier, comme dans le modèle classique en anneau du Lbest. Mais, contrairement au modèle classique qui possède des liens bidirectionnels, les particules dans Fitness sont connectées par des liens unidirectionnels, une particule ne peut donc communiquer qu'avec une seule autre particule située après elle. Toutefois, les particules sont triées par ordre décroissant, en fonction de leurs valeurs de fitness (fonction objectif), et non pas selon leurs indices prédéfinis. En effet, la particule qui a la plus grande valeur de fitness (la mauvaise particule) est placée dans la première position de l'anneau, suivie par la particule qui a la deuxième valeur, et ainsi de suite (cet arrangement a lieu en cas de minimisation).
- **Fitness-Distance-Ratio (FDR)**: est une variante de PSO se basant sur un réseau social spécial, qui a été inspiré par l'observation du comportement animal. Ce réseau social n'utilise pas une structure géométrique spécifique pour effectuer la communication entre les particules. Toutefois, il utilise une nouvelle façon de communiquer, qui est modélisée par l'ajout d'un terme à la composante sociale de l'équation (1). Dans FDR, la position et la vitesse d'une particule sont mises à jour dimension par dimension. Pour chaque dimension, la particule n'a qu'une seule informatrice. On évite ainsi que plusieurs informatrices se communiquent des informations qui s'annulent entre elles.
- **R2006**: C'est une topologie dynamique aléatoire qui a été proposée par Clerc. Elle consiste à utiliser la notion de modèle Lbest, dans lequel chaque particule est influencée par son propre voisinage défini à l'avance. Toutefois, dans R2006, la taille du voisinage est variable d'une particule à l'autre, elle peut être de n'importe quelle valeur entre 1 et S (où S est la taille de la population) avec une forte

probabilité d'être proche d'un nombre fixe k. Chaque particule choisit son voisinage aléatoirement en utilisant une distribution représentée graphiquement par une courbe ayant la forme d'une cloche non symétrique (avec une valeur de moyenne proche de k). À chaque itération, si la meilleure solution trouvée jusque-là n'a pas été améliorée, la topologie change à l'itération suivante.

d. L'Algorithme TRIBES

TRIBES est un algorithme d'optimisation par essaim particulaire sans paramètres de contrôle. Cet algorithme présente la particularité d'être totalement adaptatif, c'est-à-dire que tous les paramètres de contrôle sont calculés de manière autonome par l'algorithme. En effet, TRIBES est défini comme une boîte noire, pour laquelle l'utilisateur n'a plus aucun paramètre à régler. Il doit seulement définir le problème à résoudre (i.e. la fonction objectif, l'espace de recherche, les contraintes), ainsi que son critère d'arrêt. Cependant, il est à signaler que TRIBES ne peut pas résoudre tous les problèmes. De plus, ses résultats sont probabilistes à cause de son caractère stochastique. Le but de TRIBES, d'après son auteur, est d'être efficace dans la plupart des cas et de permettre à ses utilisateurs de gagner du temps, en évitant l'étape de réglage de la métaheuristique.
Dans TRIBES, l'essaim particulaire est divisé en plusieurs sous-essaims appelés « tribus ». Les tribus sont de tailles différentes, qui évoluent au cours du temps. Le but est d'explorer simultanément plusieurs régions de l'espace de recherche, généralement des optima locaux, avant de prendre une décision globale. Dans le but de prendre un telle décision, les tribus échangent leurs résultats tout au long du traitement. Deux types de communications sont donc à définir : la communication intra-tribu et la communication inter-tribus. Chaque tribu est composée d'un nombre variable de particules. En effet, une tribu qui peine à améliorer ses résultats génère des particules plus « exploratrices ». Les particules générées par les différentes tribus forment une nouvelle tribu, qui reste en communication avec ses génitrices. Inversement, une tribu efficace tendra à supprimer celles de ses particules qui n'ont pas contribué à sa bonne performance.
TRIBES est un algorithme compétitif, qui permet de trouver rapidement des optima locaux. Cependant, les particules ont tendance à rester dans ces optima locaux et ont du mal à en sortir.

3. Conclusion

Dans ce chapitre, nous avons présenté un état de l'art sur les métaheuristiques d'optimisation. Puis nous avons présenté quelques métaheuristiques très répandues et parmi elles, un intérêt particulier a été porté à la méthode d'optimisation par essaim particulaire (PSO). Cette méthode, qui est inspirée du monde du vivant, a rencontré un succès remarquable depuis sa création, grâce à sa simplicité. Elle présente l'avantage d'être efficace sur une vaste gamme de problèmes, sans pour autant que l'utilisateur ait à modifier la structure de base de l'algorithme. Cependant, PSO présente un problème majeur, qui rebute encore certains utilisateurs : le problème de la convergence prématurée, qui peut conduire les algorithmes de ce type à stagner dans un optimum local. Et pour finir, nous avons présenté quelques travaux qui ont été proposés pour améliorer les performances de PSO, tout en essayant de remédier à ce problème.

Table des matières

1. Introduction

La famille des métaheuristiques regroupe un ensemble de méthodes performantes en optimisation difficile. Elle offre des solutions approchées de bonne qualité pour des applications réelles de grande taille. Malgré le succès remarquable de leur démarche, les métaheuristiques présentent des difficultés, alors des essais d'amélioration ont été proposés pour obtenir la méthode la plus efficace possible. Parmi ces essais, nous pouvons citer la technique d'hybridation entre les méthodes qui permet d'arriver à des améliorations puisque les avantages et les inconvénients de chacune des méthodes hybridées se compensent. Actuellement, les métaheuristiques hybrides sont devenues très populaires car les meilleurs résultats trouvés pour plusieurs problèmes d'optimisation ont étés obtenus avec des algorithmes hybrides [3.1].

Dans ce chapitre, nous présenterons un état de l'art concernant les techniques d'hybridation et leurs différentes classifications. Puis, nous exposerons la méthode que nous avons développée pour résoudre le problème d'ordonnancement des tâches sur une grille de calcul qui est l'hybridation entre HEFT et PSO.

2. Etat de l'art de l'hybridation

2.1. Définition

L'hybridation consiste à combiner les caractéristiques de plusieurs algorithmes pour tirer profit de leurs avantages [3.2].

Les origines des algorithmes hybrides des métaheuristiques reviennent aux travaux de Glover [3.3], J. J. Grefenstette [3.4], et Mühlenbein et al. [3.5]. Mais à cette période, la plupart des chercheurs n'y accordait que peu d'intérêt. Actuellement, les métaheuristiques hybrides sont devenues plus populaires car les meilleurs résultats trouvés pour plusieurs problèmes d'optimisation combinatoires ont étés obtenus avec des algorithmes hybrides.

2.2. Classification

Plusieurs classifications ont été proposées dans la littérature. Dans ce qui suit, nous allons exposer la taxonomie proposée par Talbi dans [3.2] qui est générale et permet d'inclure facilement de nouveaux schémas d'hybridation. Dans cette classification, Talbi a divisé l'hybridation des métaheuristiques en deux grandes parties: hybridation des métaheuristiques avec des métaheuristiques et hybridation des métaheuristiques avec des méthodes exactes.

2.2.1. Hybridation métaheuristiques/métaheuristiques

L'hybridation des métaheuristiques entre elles se fait en deux classifications principales : Une classification hiérarchique et une classification à plat.

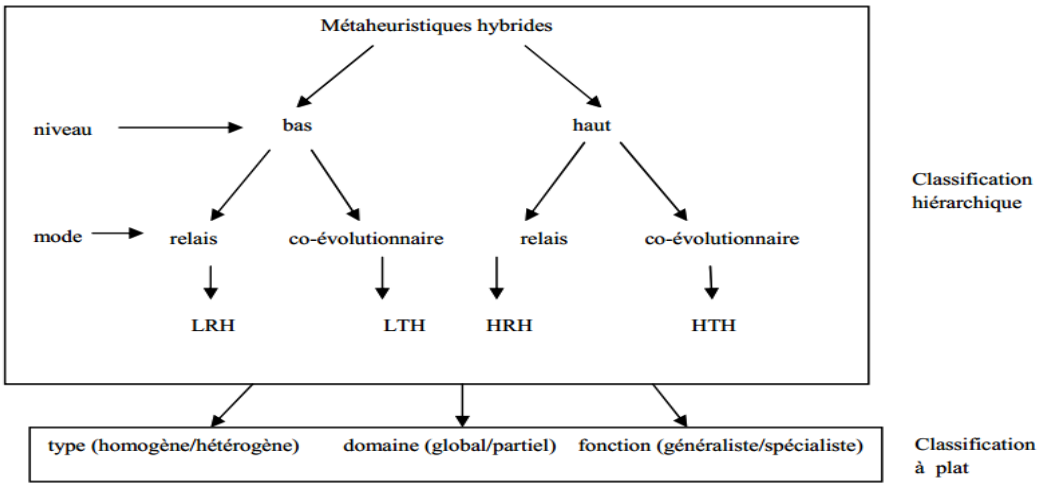


Figure 3.1 Taxonomie de l'hybridation des métaheuristiques [3.1].

a. Classification hiérarchique des métaheuristiques

Cette classification est caractérisée par le niveau et le mode de l'hybridation.

Le niveau d'hybridation peut être bas (Low-Level) ou haut (High-Level). Dans le niveau bas, une métaheuristique remplace un opérateur d'une autre méthode qui l'englobe. Par contre, dans le niveau haut de l'hybridation, chaque métaheuristique garde sa propriété au cours de l'hybridation. Chaque niveau d'hybridation engendre deux modes de coopération à savoir, le mode relais et le mode co-évolutionnaire. Dans le mode relais, les méthodes sont exécutées séquentiellement, c'est-à-dire le résultat de la première méthode est le début de la méthode suivante. Quand les différentes méthodes fonctionnent en parallèle pour explorer l'espace de recherche, on parle de mode co-évolutionnaire.

La combinaison des modes et des niveaux donne quatre classes d'hybridation qui sont l'hybridation relais de bas niveau, l'hybridation co-évolutionnaire de bas niveau, l'hybridation relais de haut niveau et l'hybridation co-évolutionnaire de haut niveau.

1. **L'hybridation relais de bas niveau** (Low-level Relay Hybrid): Elle englobe les métaheuristiques à base de solution unique dans lesquelles une autre méthode est incorporée pour former un nouvel algorithme
2. **L'hybridation co-évolutionnaire de bas niveau** (Low-level Teamwork Hybrid) : Consiste à incorporer une ou plusieurs métaheuristiques à base de solution unique dans une métaheuristique à population de solutions. L'avantage de ce type d'hybridation est de compenser la puissance d'exploitation d'une recherche locale et celle d'exploration d'une recherche globale.
3. **L'hybridation relais de haut niveau** (High-level Relay Hybrid) : consiste à utiliser les métaheuristiques de manière séquentielle c'est-à-dire la (ou les) solution(s) finale(s) de la première métaheuristique est la (ou les) solution (s) initiale(s) de la métaheuristique suivante. Dans cette procédure, toutes les méthodes gardent leur intégrité.
4. **L'hybridation co-évolutionnaire de haut niveau (**High-level Teamwork Hybrid) : Dans ce cas, les métaheuristiques utilisées travaillent en parallèle en échangeant des informations entre elles afin de trouver la solution optimale du problème posé. L'exemple le plus populaire est celui de l'algorithme génétique basé sur le modèle insulaire proposé par Tanese [3.6]. Pour cette hybridation, la population est divisée en sous-populations réparties sur les sommets d'un hypercube dont lesquels un

algorithme génétique est lancé (plusieurs algorithmes génétiques s'exécutent en parallèle). Chaque sommet correspond à une zone de recherche de solutions et périodiquement, des individus migrent entre les sommets en contribuant à trouver les solutions optimales.

b. Classification à plat des métaheuristiques

La classification à plat des métaheuristiques est caractérisée par :

- le type des méthodes hybridées,
- leur domaine d'application et
- la nature de leurs fonctions.

Selon le type d'hybridation, on trouve des méthodes hybridées homogènes où les algorithmes utilisés se basent sur la même métaheuristique comme le modèle insulaire et des méthodes hybridées hétérogènes où les métaheuristiques utilisées sont différentes.

Le domaine d'application des métaheuristiques hybridées permet de distinguer deux grandes classes d'hybridation, les hybridations globales et les hybridations partielles. L'hybridation globale a lieu lorsque toutes les méthodes hybridées sont appliquées à la totalité de l'espace de recherche. Toutes les méthodes que nous avons étudiées précédemment sont des hybridations globales. A l'opposé, l'hybridation partielle décompose un problème en sous-problèmes où chacun a son propre espace de recherche.

Selon le problème (fonction) traité, Talbi dans [3.2] distingue deux types d'hybridation, une hybridation généraliste et une hybridation spécialiste. On parle d'hybridation généraliste quand toutes les métaheuristiques hybridées traitent le même problème d'optimisation. Toutes les hybridations que nous avons citées dans la classification hiérarchique font partie de cette catégorie. A l'inverse, les hybridations spécialistes ont lieu lorsque chaque métaheuristique traite un problème différent. Un exemple de ce type est l'utilisation d'une métaheuristique pour initialiser les paramètres d'une autre métaheuristique.

2.2.2. Hybridation métaheuristiques/méthodes exactes

L'hybridation des métaheuristiques avec les méthodes exactes a été moins usuelle que l'hybridation métaheuristique/métaheuristique car la plupart des chercheurs la considérait assez inutile. En ces derniers temps, cette hybridation commence à s'étendre et un grand nombre d'articles a été publié concernant cette étude. Talbi dans [3.2] a généralisé sa taxonomie aux méthodes exactes de sorte que la classification hiérarchique est applicable à ce genre d'hybridation.

1. La classe d'hybridation relais de bas niveau (LRH) est plus efficace en ayant une métaheuristique hybridée avec une méthode exacte.
2. La classe d'hybridation co-évolutionnaire de bas niveau (LTH) regroupe une métaheuristique à base de population dont laquelle un opérateur est remplacé par une méthode exacte. La classe LTH concerne aussi toutes les métaheuristiques basées sur l'exploration du voisinage où une méthode exacte intervient pour trouver la (ou les) meilleure solution du voisinage.
3. Dans la classe d'hybridation relais de haut niveau (HRH), les métaheuristiques et les méthodes exactes hybridées sont réalisées séquentiellement en gardant leur propriété.
4. L'hybridation co-évolutionnaire de haut niveau (HTH) est difficile à réaliser entre une méthode exacte et une métaheuristique car chaque approche résout un problème différent puis, un échange d'information est indispensable entre elles. L'exécution des méthodes s'effectue en parallèle tout en gardant une communication entre les méthodes.

Pour la classification à plat, Talbi considère que les mêmes étapes qu'en hybridation métaheuristique/métaheuristique sont applicables à l'hybridation métaheuristique/méthodes exactes [3.1].

Dans [3.7], Puchinger et Raid proposent une autre classification pour l'hybridation des métaheuristiques avec les méthodes exactes. Ils divisent les méthodes hybridées en deux grandes classes : les hybridations collaboratives et celle intégratives.

- Dans **l'hybridation collaborative**, les deux approches communiquent entre elles en échangeant des informations séquentiellement ou parallèlement. *L'hybridation séquentielle* a lieu quand une méthode exacte initialise une métaheuristique ou vice-versa. Quant à *l'hybridation collaborative parallèle*, elle a lieu quand la communication entre les métaheuristiques et les méthodes exactes s'effectue parallèlement.
- Dans **l'hybridation intégrative**, l'un des algorithmes (métaheuristique ou exact) est intégré dans l'autre, c'est-à-dire qu'une métaheuristique est intégrée dans une méthode exacte ou inversement. Pour résoudre un problème combinatoire de façon exacte, une méthode exacte est incorporée dans une métaheuristique.

Remarque : La stratégie de Puchinger ressemble à celle proposée par Talbi et les deux classifications suivent les mêmes classes d'hybridation [3.1].

- Pour intégrer un algorithme dans un autre, Talbi applique l'hybridation de bas niveau (relais ou co-évolutionnaire) tandis que Puchinger applique l'hybridation intégrative.
- Pour une exécution séquentielle des méthodes, Talbi fait appel à l'hybridation relais de haut niveau et Puchinger à l'hybridation collaborative séquentielle.
    - Dans une exécution parallèle des algorithmes, on trouve l'hybridation de haut niveau co-évolutionnaire chez Talbi et l'hybridation collaborative parallèle chez Puchinger.

3. L'hybridation HEFT/DPSO

Dans cette section, nous allons présenter l'heuristique HEFT, sa variante DHEFT (Duplication-based HEFT), DPSO et enfin la méthode hybride HEFT/DPSO.

3.1. Formulation du problème d'ordonnancement

Un modèle d'un système d'ordonnancement est constitué d'une application, une plateforme cible et d'un critère de performance d'ordonnancement qui est le makespan dans notre cas.

3.1.1. Une application : est représentée par un DAG (Direct Acyclic Graph), G= (V, E), où V est l'ensemble de *v* tâches et E est l'ensemble de e arêtes entre les tâches. Une arête (i, j)∈ E représente une contrainte de dépendance telle que la tâche n_i doit être exécutée avant que la tâche n_j soit lancée. Une tâche sans parents est appelée tâche d'entrée. Une tâche sans successeurs est appelée tâche de sortie. La matrice *Données* de dimension *v x v* représente les données à communiquer entre les tâches. $Données_{i,k}$ est le volume de données à transférer depuis la tâche n_i vers la tâche n_k **.** La figure suivante montre une application avec 10 tâches représentée par un DAG.

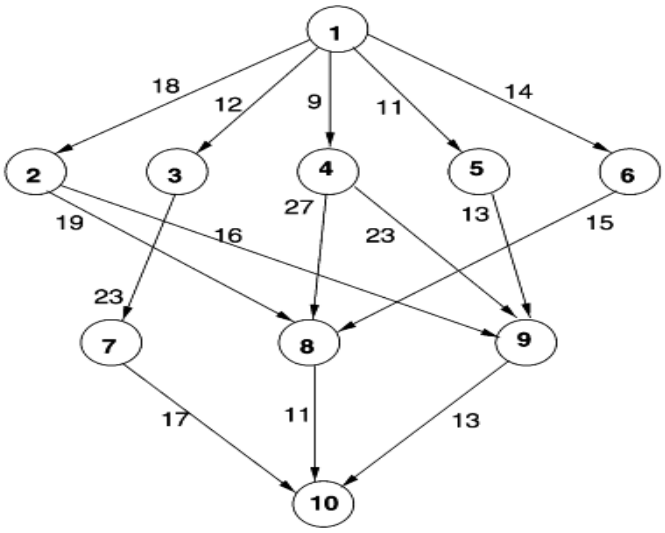


***Figure 3.2*:** Une application représentée par un DAG [3.8].

Le tableau suivant donne le volume de données échangées entre les tâches de l'application représentée dans la figure 3.2. Les valeurs '-1' veux dire qu'il n'y a pas d'échange de données entre les tâches concernées. L'unité de ces mesures est l'unité de volume de données qui peut être par exemple l'octet.

Tâche / Tâche	1	2	3	4	5	6	7	8	9	10
1	-1	18	12	9	11	14	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1	19	16	-1
3	-1	-1	-1	-1	-1	-1	23	-1	-1	-1
4	-1	-1	-1	-1	-1	-1	-1	27	23	-1
5	-1	-1	-1	-1	-1	-1	-1	-1	13	-1
6	-1	-1	-1	-1	-1	-1	-1	15	-1	-1
7	-1	-1	-1	-1	-1	-1	-1	-1	-1	17
8	-1	-1	-1	-1	-1	-1	-1	-1	-1	11
9	-1	-1	-1	-1	-1	-1	-1	-1	-1	13
10	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Tableau 3.1: Matrice des données.

3.1.2. Une plateforme : est constituée d'un ensemble de *q* machines hétérogènes complètement connectés entre elles. Nous supposons que les communications entre machines sont réalisées sans contention et avec recouvrement de calcul. En plus la préemption n'est pas permise. La matrice *Coût_Calcul* de dimension *v x q* représente les coûts de calcul des tâches sur les machines. $Coût\_Calcul_{i,j}$ donne le temps estimé de l'exécution de la tâche n_i sur la machine m_j . Le coût de moyen de l'exécution d'une tâche n_i est :

$$\overline{Coût\_Calcul_i} = \frac{\sum_{j=1}^{q} Coût\_Calcul_{i,j}}{q}.$$

Le tableau suivant donne les coûts de calcul des tâches de l'application représentée dans la figure 3.2 sur une plateforme de trois machines. Ces mesures sont en unité de temps (ça peut être en milliseconde comme en heure):

Machine / Tâche	m_1	m_2	m_3
1	14	16	9
2	13	19	18
3	11	13	19
4	13	8	17
5	12	13	10
6	13	16	9
7	7	15	11
8	5	11	14
9	18	12	20
10	21	7	16

Tableau 3.2: La matrice des coûts de calcul.

Les vitesses de transfert entre les machines sont sauvegardées dans la matrice $Vitesse\_transfert$ de dimension $q \; x \; q$. La vitesse entre une machine et elle-même est définit comme nulle. Dans le tableau suivant qui donne les vitesses de transferts entre les trois machines d'une plateforme, l'unité de mesure est l'unité de volume de données sur l'unité de temps.

Machine / Machine	m_1	m_2	m_3
m_1	0	1	1
m_2	1	0	1
m_3	1	1	0

Tableau 3.3: Matrice de vitesse de transfert entre les machines.

Le coût de communication de l'arête (i, j), qui est pour le transfert de données depuis la tâche n_i (ordonnancée sur m_p) vers la tâche n_j (ordonnancée sur m_k), est définit par :

$$C_{i,j} = \frac{Données_{i,j}}{Vitesse\_transfert_{p,k}} .$$

Le coût moyen de communication d'une arête (i, j) est :

$$\overline{C_{i,j}} = \frac{Données_{i,j}}{\overline{Vitesse\_transfert}} .$$

Où $\overline{Vitesse\_transfert}$ est la moyenne des vitesses de transfert entre les machines.
Quand la tâche n_i et n_j sont ordonnancées sur la même machine alors $C_{i,j}=0$.

3.1.3. makespan (temps de complétition): pour calculer le makespan, il est nécessaire de définir les attributs suivants :

1. EST(n_i, m_j) 'Earliest execution Start Time' de la tâche n_i sur la machine m_j.

- Pour la tâche d'entrée $n_{entrée}$: EST($n_{entrée}$, m_j)=0.
- Pour les autres tâches : *EST*(n_i, m_j) =max $\{avail[j],\ \max_{n_m \in pred\ (n_i)}(AFT(n_m) + c_{m,i})\}$

Où : pred(n_i) est un ensemble des prédécesseurs immédiats de la tâche n_i . *avail*[j] est le prochain temps où la machine m_j sera prête pour l'exécution des tâches (la machine sera disponible). Le 'max' interne de l'équation de EST donne le temps où toutes les données nécessaires à l'exécution de n_i sont arrivées à la machine m_j.

2. *EFT*(n_i, m_j) 'Earliest execution Finish Time' de la tâche n_i sur la machine m_j**.**

- Pour toute les tâches : *EFT*(n_i, m_j)= $Coût\_Calcul_{i,j}$ + *EFT*(n_i, m_j).

Après qu'une tâche n_i soit ordonnancée sur une machine m_j alors :

- 'Actual Start Time' de la tâche n_i est : AST(n_i) = EST(n_i, m_j) et
- 'Actual Finish Time' de la tâche n_i est : AFT(n_i) = EFT(n_i, m_j).

Après l'ordonnancement de toutes les tâches du DAG, le temps de complétition de toutes les tâches (makespan) est l'Actual Finish Time de la tâche de sortie.

makespan= AFT(n_{sortie}) .

L'objectif du problème d'ordonnancement de tâches est de déterminer un assignement des tâches d'une application donnée sur les machines de la plateforme de manière à minimiser le makespan.

3.2 HEFT

HEFT [3.8] est parmi les algorithmes d'ordonnancements de liste les plus répandus. Il détermine un ordonnancement totalement statique d'un DAG sur un environnement hétérogène de manière à minimiser la durée totale d'exécution de l'application (makespan).

3.2.1. L'attribut du graphe utilisé par HEFT

Les tâches sont triées par leurs priorités d'ordonnancement qui sont basées sur le rang upward ($rank_u$). $rank_u$ est la distance maximale d'une tâche par rapport à tâche de sortie du DAG. Il est défini pour la tâche n_i récursivement comme suit:

$$rank_u(n_i) = \overline{Coût\_Calcul_i} + \max_{n_j \in succ\ (n_i)}(\overline{C_{i,j}} + rank_u(n_j))$$

où :

- succ(n_i) est l'ensemble des successeurs immédiats de la tâche n_i.
- $\overline{Coût\_Calcul_i}$ le coût moyen de l'exécution de la tâche n_i.
- $\overline{C_{i,j}}$ est le coût moyen des communications de l'arête (i, j).

Il est calculé pour toutes les tâches du DAG en commençant par la tâche de sortie qui a comme rang :

$rank_u(n_{sortie}) = \overline{Coût\_Calcul_{sortie}}$.

Le tableau suivant donne les $rank_u$ des tâches de l'application de la figure 3.2 en utilisant les valeurs de la matrice de coûts de calcul donné dans le tableau 3.2 :

Tâche	1	2	3	4	5	6	7	8	9	10
$rank_u$	108	77	80	80	69	63.33	42.667	35.667	44.333	14.667

Tableau 3.4 : Les valeurs de rang pour les tâches.

3.2.2. Les phases de l'algorithme HEFT

L'algorithme HEFT a deux phases d'exécution

1. La phase de prioritisation : les priorités de toutes les tâches sont calculées en utilisant $rank_u$. Ensuite une liste de tâches est générée en triant les tâches par leurs $rank_u$ (priorités) décroissants.
2. La phase de sélection de machine : dans cette phase, les tâches sont assignées aux machines qui minimisent leurs EFT.

1. **Définir** le coût de calcul pour chaque tâche et les coûts de communication.
2. **Calculer** le rang pour toutes les tâches en parcourant le DAG vers le haut à partir de puits.
3. **Trier** les tâches dans une liste d'ordonnancement par le rang dans un ordre décroissant.
4. **Tant qu**'il y'a des tâches non ordonnées dans la liste
   Sélectionner la première tâche 'T' dans la liste des tâches
5. **Pour** toutes les machines m_j avec j = 1..q
   Calculer EFT (T, m_j)
6. **Assigner** la tâche 'T'à la machine M_j qui a le EFT minimum
7. **Fin** de tant que.

Figure 3.3: L'algorithme HEFT.

Le makespan de l'application représentée par la figure 3.2 en utilisant les valeurs des tableaux 3.1, 3.2 et 3.3 avec la méthode HEFT égale à 93 (unité de temps).

3.2.3. Duplication-based HEFT (DHEFT)

DHEFT [3.9] est une variante de HEFT qui prend en charge la technique de duplication des tâches. Son principe général est le même que celui de HEFT. L'amélioration apportée est dans le fait que quand le temps de communication est plus grand comparé au temps de calcul, une tâche T_j qui a été ordonnancée sur d'autres machines, est dupliquée sur la machine P_m pour l'exécuter et fournir les données produites par cette tâches aux autres qui en sont dépendantes.

Ainsi le temps de communication des données sera supprimé car les tâches sont assignées à la même machine, donc le temps de communication des données est nulle.

Le makespan de l'application représentée par la figure 3.2 en utilisant les valeurs des tableaux 3.1, 3.2 et 3.3 avec la méthode DHEFT égale à 90 (unité de temps).

3.3. DPSO (Discete Particle Swarm Optimization)

Dans le chapitre précédent, nous avons décrit l'algorithme d'optimisation par essaim particulaire pour les problèmes d'optimisations ayant des solutions dans le domaine continu. Dans ce qui suit, nous allons présenter une version discrète de PSO adaptée au problème d'ordonnancement appelée *DPSO (Discete Particle Swarm Optimization)* [3.10].

3.3.1. Discrete Particle Swarm Optimization (DPSO)

Nous utilisons une version discrète de PSO adaptée au problème d'ordonnancement. Chaque particule k dans l'essaim est modélisée par :

- Une position qui doit représenter la séquence des tâches qui doivent être exécuté sur les machines (l'ordonnancement): $\overrightarrow{X}_k = (x_{k1}, x_{k2}, \ldots, x_{kD})$.
- Une vitesse qui doit être adaptée au problème d'ordonnancement : $\overrightarrow{V}_k = (v_{k1}, v_{k2}, \ldots, v_{kD})$.
- Une meilleure position par laquelle elle est déjà passée : $\overrightarrow{Pbest}_k = (pbest_{k1}, pbest_{k2}, \ldots, pbest_{kD})$.
- Comme nous avons choisi d'utiliser la version globale de PSO, alors la meilleure position atteinte par l'ensemble des particules de l'essaim : $\overrightarrow{Gbest} = (gbest_1, gbest_2, \ldots, gbest_D)$.
- Une fitness qui est égale au makespan de l'ordonnancement que représente la position actuelle de la particule.

a. La position d'une particule

La représentation qui est une étape clé dans la réalisation d'un bon algorithme de PSO, est la correspondance entre la solution du problème (l'ordonnancement) et la particule de PSO. On peut distinguer deux représentations : directe et indirecte.

1) *La représentation directe :* Les solutions sont représentées dans un vecteur *X* de dimension *1xn* où *n* est le nombre du tâches de l'application à ordonnancer. Les éléments de ce vecteur sont des entiers naturels compris dans l'intervalle [0, *m* [où *m* est le nombre du machines. La position d'une particule i est le vecteur X_i.

X_i[j] : est l'indice de la machine pour laquelle la tâche *j* est assignée.

2) *La représentation indirecte :* Les solutions sont représentées dans une matrice *X* de dimension *m x n* où *m* est le nombre de machines et *n* celui de tâches. Cette matrice *X* a deux propriétés :
   a) X [i, j] $\in\{0, 1\}$, $\forall$ i$\in\{1, 2, ..., m\}$ et $\forall$ j $\in\{1, 2, ..., n\}$.
   b) dans chaque colonne de X seul un élément est à 1 les autres sont à 0.

La position d'une particule k est la matrice X_k. Si X_k[i, j] = 1 alors la tâche *j* est assignée à la machine *i.*
Dans la matrice de position, chaque colonne représente l'allocation d'une tâche et chaque ligne représente les tâches allouées à une machine.

Exemple : Soit une application avec 5 tâches à exécuter sur 3 machines.
La figure suivante montre la représentation directe et indirecte d'une position potentielle d'une particule.

T_1	T_2	T_3	T_4	T_5
3	1	2	1	2

La représentation directe

	T_1	T_2	T_3	T_4	T_5
M_1	0	1	0	1	0
M_2	0	0	1	0	1
M_3	1	0	0	0	0

La représentation indirecte

Figure 3.4 : La représentation directe et indirecte d'une position.

Suivant cette figure, les tâches T_2 et T_4 seront exécutées sur la machine M_1, T_3 et T_5 sur la machine M_2, et T_1 sur M_3.

Remarque : Puisque Pbest et Gbest sont des positions alors leurs représentations suivra automatiquement celle de la position d'une particule.

b. La vitesse d'une particule

La vitesse d'une particule k est une matrice de dimension *m x n* où *m* est le nombre de machines et *n* celui de tâches. Tel que : V_k[i, j]$\in$ $[-V_{max}, V_{max}]$, $\forall$ i$\in${1, 2, ..., *m*} et $\forall$ j $\in${1, 2, ..., *n*}. Elle est initialisée aléatoirement au lancement de programme pour chaque particule. La vitesse est toujours représentée de cette manière pour les deux représentations des positions (directe et indirecte).

c. La fitness d'une particule

La fitness d'une particule est égale au makespan de l'ordonnancement représenté par la position de la particule. Le makespan est calculé par la méthode exposée dans le paragraphe 3.1.3.

Fitness=makespan.

d. Déplacement d'une particule

La modélisation du mécanisme de déplacement d'une particule dans l'espace de recherche est influencée par la représentation utilisée pour modéliser la position d'une particule. Et pour bien le comprendre, nous allons présenter d'abord le déplacement d'une particule ayant comme représentation indirecte pour sa position.

1. La représentation indirecte

La mise à jour de la matrice vitesse d'une particule k, se fait comme suit :

$$V_k^{t+1}[i, j]=w \cdot V_k^t[i, j] + c_1 \cdot r_1 \cdot (pbest_k^t[i, j]-X_k^t[i,j]) + c_2 \cdot r_2 \cdot (gbest_k^t[i, j]-X_k^t[i, j]) \quad (1).$$

Où:

- V_k^t est la vitesse actuelle (de l'itération en cours).
- V_k^{t+1} est la prochaine vitesse de l'itération suivante.
- w est le coefficient d'inertie.
- c_1 et c_2 sont les coefficients d'accélération.
- r_1 et r_2 sont deux nombres aléatoires tirés uniformément dans$[0,1]$ à chaque itération *t*.
- X_k^t est la matrice de position d'une particule k.
- $pbest_k^t$ est la matrice de la meilleure position par laquelle la particule k est déjà passée.
- $c_1.r_1$.($pbest_k^t$[i, j]-X_k^t[i, j]) est la composante cognitive.
- $gbest_k^t$ est la meilleure position atteinte par les particules de l'essaim.
- $c_2.r_2$. ($gbest_k^t$[i, j]-X_k^t[i, j]) est la composante sociale.

La mise à jour de la matrice position de la particule k, se fait comme suit :

$$X_k^{t+1}[i,j]=\begin{cases}1 & si\ V_k^{t+1}[i,j]=\max\{V_k^{t+1}[i,j]\}, \forall\ i \in \{1,2,\dots,m\}\\ 0 & sinon\end{cases} \quad (2).$$

Où :

- X_k^{t+1}est la nouvelle position de la particule k.
- V_k^{t+1} est ka nouvelle vitesse calculée par l'équation (1).

L'équation (2) veut dire que pour chaque colonne de la matrice position X, la valeur 1 est affectée à l'élément qui correspond à l'élément de la matrice vitesse V qui a la valeur maximale. Les autres éléments de la matrice position sont à zéro.

NB : Dans une matrice de vitesse, si une colonne à plus d'un élément qui à la même valeur maximale (c.à.d. plusieurs machines candidates) alors on choisi la machine qui minimise le temps de calcul pour la tâche correspondante.

2. La représentation directe

La mise à jour de la matrice vitesse d'une particule k, se fait comme suit :

Dans la représentation indirecte, les valeurs que peut prendre X_k^t[i, j] et $pbest_k^t$[i, j] sont soit 0 ou 1.

Donc les valeurs que peut avoir $\boldsymbol{c_1.r_1}$.($\boldsymbol{pbest_k^t}$**[i, j]**-$\boldsymbol{X_k^t}$**[i, j])** (la composante cognitive) dans l'équation (1) peuvent soit être :

- **0**, si $pbest_k^t$[i, j]=X_k^t[i, j]
- **+** ($\boldsymbol{c_1.r_1}$), si $pbest_k^t$[i, j]=1 et X_k^t[i, j]=0, ou
- **-** **($\boldsymbol{c_1.r_1}$)**, si $pbest_k^t$[i, j]=0 et X_k^t[i, j]=1.

Soit : $pbest_k^t$[j]=z et X_k^t[j] =q avec z≠ q

Cela veut dire que[1] $\begin{cases} pbest_k^t[z,j]=1 \\ X_k^t[q,j]=1 \end{cases}$ et comme z≠ q $alors$[2] $\begin{cases} pbest_k^t[q,j]=0 \\ X_k^t[z,j]=0 \end{cases}$

donc :

$$\begin{cases} V_k^{t+1}[z,j] = V_k^t[z,j] + (\boldsymbol{c_1}.\boldsymbol{r_1}) \text{ car : } \boldsymbol{pbest_k^t[z,j] - X_k^t[z,j] = +1} \\ V_k^{t+1}[q,j] = V_k^t[q,j] - (\boldsymbol{c_1}.\boldsymbol{r_1}) \text{ car : } \boldsymbol{pbest_k^t[q,j] - X_k^t[q,j] = -1} \end{cases}$$

On suit le même raisonnement avec $X_k^t[i,j]$ et $gbest_k^t[i,j]$ pour la composante sociale de l'équation (1).

La mise à jour du vecteur position de la particule k se fait comme suit :

Après la mise à jour de la vitesse, la nouvelle position de la particule est obtenue par la formule :

$X_k^t[j]=\varphi$ si $V_k^t[\varphi,j] = \max\{V_k^t[i,j]\}$. (3)

$\forall j \in (1, 2, ..., n)$ $\forall i \in (1, 2, ..., m)$,

c.à.d. que pour chaque élément du vecteur de position, une machine de la colonne correspondante dans la matrice de vitesse est choisie s'il a une valeur maximale dans cette colonne.

NB : Comme dans la représentation directe, dans une matrice de vitesse, si une colonne à plus d'un élément qui à la même valeur maximale (c.à.d. plusieurs machines candidates) alors on choisi la machine qui minimise le temps de calcul pour la tâche correspondante.

Exemple : Pour le problème d'ordonnancement de 5 tâches sur 3 machines.

Soit X_k^t= (1, 2, 2, 1,3), $pbest_k^t$= (2, 2, 1, 3, 2), w=1, c_1= 2, c_2= 0, et r_1= r_2= 1. Et soit la matrice vitesse V_k^t suivante :

	T_1	T_2	T_3	T_4	T_5
M_1	4	-2	4	4	-3
M_2	1	3	5	0	-2
M_3	2	-1	-3	-3	1

Mise à jour de la matrice vitesse :

Pour la tâche T_1: $pbest_k^t[1]$=2 et $X_k^t[1]$ =1 avec 2≠ 1

Donc : $\begin{cases} V_k^{t+1}[2,1] = V_k^t[2,1] + (c_1.r_1) = 1 + 2 = 3; \\ V_k^{t+1}[1,1] = V_k^t[1,1] - (c_1.r_1) = 4 - 2 = 2; \end{cases}$

Et $V_k^{t+1}[3,1]$=$V_k^t[3,1] = 2$ (reste inchangé).

[1] $pbest_k^t[j]$=z de la représentation directe correspond à $pbest_k^t[z,j]$=1 dans la représentation indirecte et $X_k^t[j] = q$ correspond, dans la représentation indirecte à $X_k^t[j,j]$=1.

[2] Dans la représentation indirecte, sur une colonne il ne peut y avoir qu'un seul élément à 1 les autres sont à 0.

La matrice vitesse devient

	T_1	T_2	T_3	T_4	T_5
M_1	**2**	-2	4	4	-3
M_2	**3**	3	5	0	-2
M_3	2	-1	-3	-3	1

Pour la tâche T_2: $pbest_k^t[2]$=2 et $X_k^t[2]$ =2 avec 2= 2 donc ne faire rien.

Pour la tâche T_3: $pbest_k^t[3]$=1 et $X_k^t[3]$ =2 avec 1≠ 2

Donc : $\begin{cases} V_k^{t+1}[1,3] = V_k^t[1,3]+(c_1.r_1) = 4+2 = 6; \\ V_k^{t+1}[2,3] = V_k^t[2,3] - (c_1.r_1) = 5-2 = 3; \end{cases}$

Et $V_k^{t+1}[3,3]$=$V_k^t[3,3] = -3$ (reste inchangé).

La matrice vitesse devient :

	T_1	T_2	T_3	T_4	T_5
M_1	**2**	-2	**6**	4	-3
M_2	**3**	3	**3**	0	-2
M_3	2	-1	-3	-3	1

Pour la tâche T_4: $pbest_k^t[4]$=3 et $X_k^t[4]$ =1 avec 3 ≠ 1

Donc : $\begin{cases} V_k^{t+1}[3,4] = V_k^t[3,4]+(c_1.r_1) = -3+2 = -1; \\ V_k^{t+1}[1,4] = V_k^t[1,4] - (c_1.r_1) = 4-2 = 2; \end{cases}$

Et $V_k^{t+1}[2,4]$=$V_k^t[2,4] = 0$ (reste inchangé).

La matrice vitesse devient :

	T_1	T_2	T_3	T_4	T_5
M_1	**2**	-2	**6**	**2**	-3
M_2	**3**	3	**3**	0	-2
M_3	2	-1	-3	**-1**	1

Pour la tâche T_5: $pbest_k^t[5]$=2 et $X_k^t[5]$ =3 avec 2≠ 3

Donc : $\begin{cases} V_k^{t+1}[2,5] = V_k^t[2,5]+(c_1.r_1) = -2+2 = 0; \\ V_k^{t+1}[3,5] = V_k^t[3,5] - (c_1.r_1) = 1-2 = -1; \end{cases}$

Et $V_k^{t+1}[1,5]$=$V_k^t[1,5] = -3$ (reste inchangé).

La matrice vitesse devient :

	T_1	T_2	T_3	T_4	T_5
M_1	**2**	-2	**6**	**2**	-3
M_2	**3**	3	**3**	0	**0**
M_3	2	-1	-3	**-1**	**-1**

La mise à jour du vecteur position : se fait comme suit pour la tâche :

T_1 : max $\{V_k^t[1,1], V_k^t[2,1], V_k^t[3,1]\}$= $V_k^t[2,1]$ alors $X_k^t[1]$=2,

T_2 : max $\{V_k^t[1,2], V_k^t[2,2], V_k^t[3,2]\}$= $V_k^t[2,2]$ alors $X_k^t[2]$=2,

T_3 : max $\{V_k^t[1,3], V_k^t[2,3], V_k^t[3,3]\}$= $V_k^t[1,3]$ alors $X_k^t[3]$=1,

T_4 : max $\{V_k^t[1,4], V_k^t[2,4], V_k^t[3,4]\}$= $V_k^t[1,4]$ alors $X_k^t[4]$=1,

T_5 : max $\{V_k^t[1,5], V_k^t[2,5], V_k^t[3,5]\}$= $V_k^t[2,5]$ alors $X_k^t[5]$=2,

Alors $\boldsymbol{X_k^{t+1}}$**= (2, 2, 1, 1,2).**

Comme nous pouvons le constater, la particule k c'est approchée de $pbest_k^t$. Et si le processus de déplacement se répète encore une fois alors la position de la particule k sera la même que $pbest_k^t$ (X_k^{t+2}=$pbest_k^t$) car c_2=0 donc la composante sociale n'est pas prise en compte et le déplacement de la particule se fait en direction de $pbest_k^t$ seulement.

La mise à jour de la matrice vitesse :

Pour la tâche T_1, T_2, T_3, T_5: $pbest_k^t[2]$=$X_k^{t+1}[2]$ donc ne faire rien.

Pour la tâche T_4: $pbest_k^t[4]$=3 et $X_k^{t+1}[4]$ =1 avec $3 \neq 1$

Donc : $\begin{cases} V_k^{t+1}[3,4] = V_k^t[3,4]+(c_1.r_1) = -1+2 = 1 \\ V_k^{t+1}[1,4] = V_k^t[1,4] - (c_1.r_1) = 2-2 = 0; \end{cases}$

La matrice vitesse devient :

	T_1	T_2	T_3	T_4	T_5
M_1	2	-2	6	**0**	-3
M_2	3	3	3	0	0
M_3	2	-1	-3	**1**	-1

La mise à jour du vecteur position : seule la tâche T_4 changera de position :

T_4 : max $\{V_k^t[1, 4], V_k^t[2, 4], V_k^t[3, 4]\} = V_k^t[3, 4]$ alors $X_k^t[4]=3$.

Alors $\boldsymbol{X_k^{t+2}}$**= (2, 2, 1, 3,2)=** $pbest_k^t$.

L'algorithme suivant résume la procédure de déplacement des particules (mise à jour des vitesses et positions) dans le cas de l'utilisation de la représentation directe.

POUR chaque particule k = 1, ..., P **faire**
  Pour chaque tâche j = 1, ..., n **faire** //mise à jour de la matrice vitesse.
    q = $X_k^t[j]$;
    z = $pbest_k^t[j]$;
    s = $pbest_k^t[j]$;
    Si q ≠z **alors**
      $V_k^t[q, j]$ =w. $V_k^t[q, j]$ - c1 × r1;
      $V_k^t[z, j]$ w. . $V_k^t[z, j]$ + c1 × r1;
    fin
    Si q ≠s **alors**
      $V_k^t[q, j]$ =w. $V_k^t[q, j]$ - c1 × r1;
      $V_k^t[s, j]$ =w. $V_k^t[s, j]$ - c1 × r1;
    fin
  fin

  Pour chaque tâche j = 1, ..., n **faire** //mise à jour du vecteur position.
    Si $V_k^t[\varphi, j] = \max \{V_k^t[i, j]\}$ ∀ i ∈ (1, 2, ..., *m*) **alors**
      $X_k^t[j] = \varphi$;
    fin
  fin
FIN

Figure3.5 : Procédure de déplacement d'une particule.

e. Le coefficient d'inertie

Nous avons choisi de faire varier le coefficient d'inertie w linéairement avec le temps selon la formule suivante :

$w = w_{min} + (w_{max} - w_{min}).(iter/max_{iter})$ (5)

Où :

- iter est l'itération courante.
- max_{iter} est le nombre maximal d'itérations.
- w_{max} et w_{min} désignent respectivement les valeurs maximum et minimum du coefficient w (généralement, w_{max} , w_{min} ∈[0, 1]).

f. L'algorithme de DPSO implémenté

Nous avons choisi d'implémenter DPSO en utilisant la représentation directe des positions puisque cette dernière consomme moins d'espace mémoire que la représentation indirecte. Aussi nous avons choisi d'utiliser la version globale de PSO.

Génération et initialisation aléatoire des particules (positions et vitesses aléatoires);
Tanque ($iter < max_{iter}$) **faire** // nombre maximal d'itération n'est pas atteint.
  Pour chaque particule k = 1, . . ., P **faire**
    Si Fitness (X_k) > Fitness ($pbest_k$) **alors**
      $pbest_k = X_k$;
    fin
    Si Fitness ($pbest_k$) > Fitness ($gbest_k$) **alors**
      $gbest = pbest_k$;
    fin
  fin
  Pour chaque particule k = 1, . . ., P **faire**
    Déplacement de la particule k ; //suivant l'algorithme de déplacement d'une particule.
  fin
  iter =*iter*+1;
FIN tanque

Figure 3.6: L'Algorithme de DPSO.

Le makespan de l'application représentée par la figure 3.2 en utilisant les valeurs des tableaux 3.1, 3.2 et 3.3 avec la méthode DPSO égale à 93 (unité de temps).

3.4. L'hybridation HEFT/DPSO

Notre objectif est d'améliorer les performances de PSO qui présente l'inconvénient de converger rapidement vers des solutions prématurées (minimums locaux). Pour cela nous proposons d'hybrider la version discrète de PSO avec l'algorithme HEFT (Heterogeneous Earliest Finish Time), qui est un bon algorithme d'ordonnancement de tâches dépendantes de type liste, facile à implémenter.

L'organigramme de la figure 3.7 illustre la méthode que nous avons suivie pour l'hybridation HEFT/DPSO. Au lieu de générer aléatoirement la population initiale de particules de l'algorithme PSO, dans notre travail, nous proposons d'améliorer cette étape en mettant en œuvre l'algorithme HEFT : ce dernier génère une solution avec un bon makespan. Ensuite, cette solution est injectée dans la population initiale de PSO et considérée comme une particule. Les autres particules de l'essaim sont générées aléatoirement. Toutes les particules sont ensuite affinées en les faisant évoluer grâce à l'algorithme PSO. Cette hybridation devra permettre d'améliorer conjointement l'efficacité de HEFT et les performances de PSO.

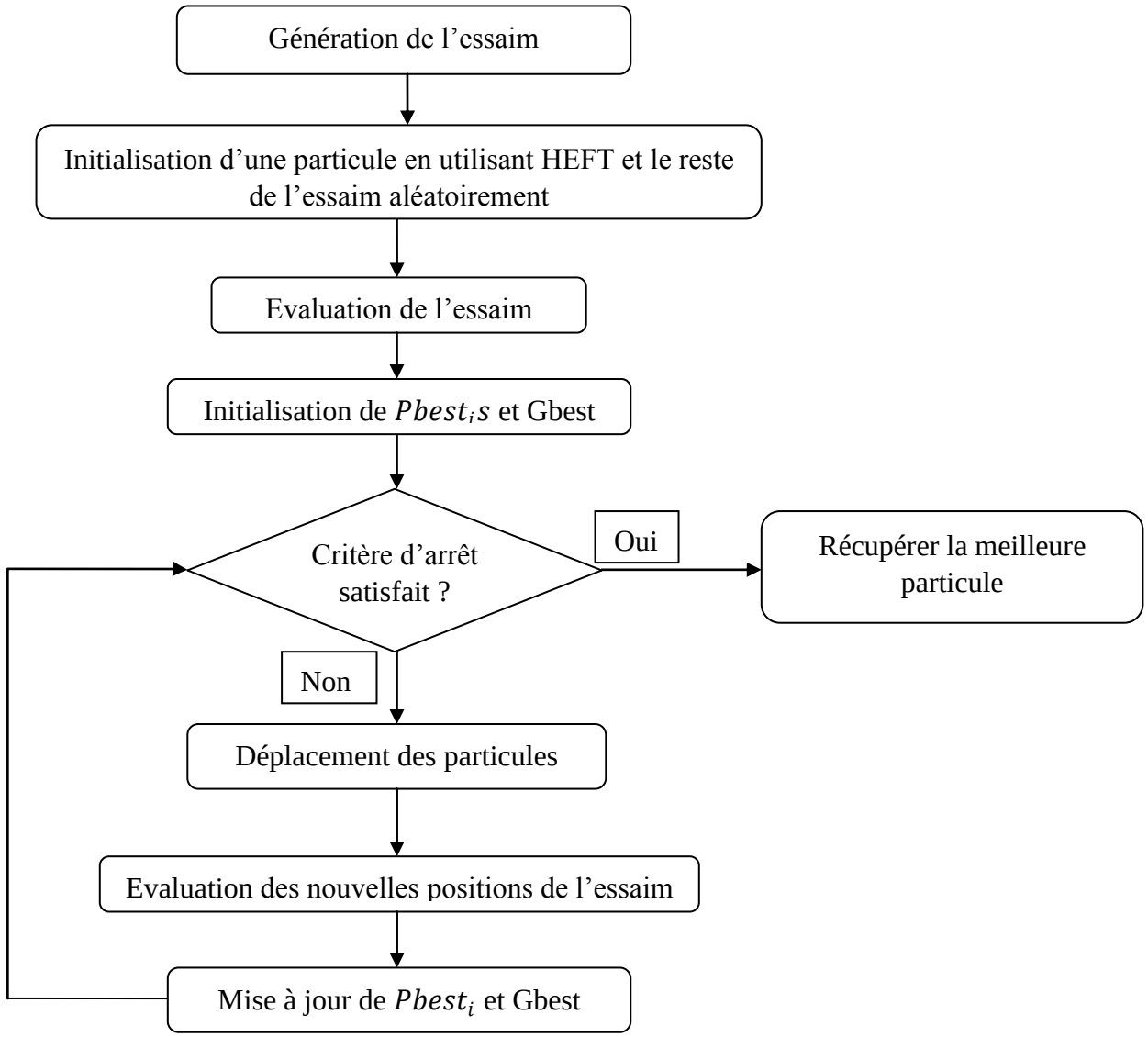


Figure 3.7: L'organigramme de l'hybridation HEFT/DPSO.

4. Conclusion

Dans ce chapitre, nous avons présenté les techniques d'hybridation des métaheuristiques. Ensuite, nous avons exposé la formulation du problème d'ordonnancement et donné des détails sur l'heuristique HEFT, sa version DHEFT et DPDO. Puis, nous avons proposé une hybridation de la méthode DPSO avec HEFT.

Dans le chapitre suivant, nous allons tester notre méthode pour vérifier ses performances.

Table des matières

1. Introduction

Dans ce chapitre, nous présenterons l'application que nous avons implémentée, puis nous effectuerons une étude comparative avec des algorithmes déjà implémentés dans le but d'évaluer notre travail.

2. Implémentation

Après avoir décrit dans le chapitre précédent l'adaptation de PSO au problème d'ordonnancement dans les grilles informatique (DPSO) et l'hybridation HEFT/DPSO, nous allons, dans cette partie, décrire la réalisation de l'hybridation dans un environnement orienté objet (C++). Pour cela dans cette partie, nous commençons par présenter l'environnement et les outils utilisés pour le développement de notre application, puis les modules implémentés et en fin quelques interfaces de notre logiciel.

2.1. Environnement et outils de développement

2.1.1. Description du langage C++

Pour réaliser notre application nous avons utilisé le langage C++. C++ est un langage de programmation crée par Bjarne Stroustrup et normalisé en 1998. Il est une amélioration de langage C. Il apporte notamment la programmation orienté objet, la gestion des exceptions, la gestion des références qui remplacent partiellement l'usage des pointeurs, la surcharge des operateurs et les templates. Une rétrocompatibilité a été gardée avec le C (les programmes C compilent sans difficulté avec un compilateur C++) [4.1].

La programmation orienté objet(POO) est un type de programmation qui a pour avantage de posséder une meilleure organisation, surtout pout les gros programmes qui seront agencés de façon plus logique et seront donc plus facilement modifiables. Parmi ses notions utilisés, nous citons **l'encapsulation** pour sécurisé le programme contre les fausses manœuvres, le **polymorphisme** qui vient du Grec et signifie « plusieurs forme »et **l'héritage** [4.2].

2.1.2. L'environnement de développement

De nombreux environnements de développement sont consacrés à C et C++, nous citons parmi eux :

- Code ::Blocks;
- Dev C++ ;
- Visual C++.

En ce qui nous concerne, nous avons choisi Code ::Blocks qui est un environnement de développement libre et multi-plateforme (disponible sous Windows, Linux et Mac OS X).

2.1.3. Les outils de développement

2.1.3.1. La bibliothèque QT

Pour réaliser l'interface graphique de notre application, nous avons choisi la bibliothèque Qt. Qt est un framework (un ensemble de modules), multiplateformes développé par la société « Trolltech » qui a été créée à la base pour réaliser les interfaces graphiques. Elle propose une double licence :

- La version OpenSource de Qt (libre et gratuite): si le programme développé avec Qt est libre c.à.d. le code source est diffusé.
- La version commerciale: si le programme développé est propriétaire c.à.d. le code source n'est pas diffusé.

Parmi les fonctionnalités offertes par Qt, sous forme de modules, nous pouvons citer :

- **Module GUI** : la partie consacrer pour la création de fenêtres.
- **Module OpenGL** : pour ouvrir une fenêtre contenant de la 3D.
- **Module de dessin** : pour dessiner dans les fenêtres (en 2D).
- **Module réseau** : ensemble d'outils pour accéder au réseau.
- **Module SVG** : pour créer des images et animations vectorielles, à la manière de Flash.
- **Module de script** : Qt supporte le Javascript (ou ECMAScript), qui peut être réutilisé dans des applications pour ajouter des fonctionnalités, sous forme de plugins par exemple.
- **Module XML** : pour l'échanger des données avec des fichiers formés à l'aide de balises (comme le XHTML).
- **Module SQL** : qui permet un accès aux bases de données (MySQL, Oracle, PostgreSQL...).

2.1.3.2. MinGW (Minimalist GNU for Windows)

Pour l'installation de Qt sous Windows, il faut d'abord avoir la bonne version du **MinGW** qui est une adaptation des logiciels de développement et de compilation du **GNU**[1] (GCC – GNU Compiler Collection), à la plate-forme **Win32**[2]**.** Il est sous la licence libre.

Remarque : Nous avons installé la version **Qt-4.8.2** qui peut être téléchargé depuis [4.3] en choisissant l'exécutable « qt-win-opensource-4.8.2-mingw », avec le compilateur « mingw 4.4.0 with pythongdb » téléchargeable depuis [4.4] et nous avons suivi les étapes données dans [4.5] pour la compilation et l'exécution des projets Qt.

2.2. Implémentation de l'application

Notre application peut être séparée en deux parties :

1. Le contrôleur : c'est la partie essentielle de notre application qui génère l'ordonnancement. Il est constitué de quatre classes : **LectureFichier** qui récupère les données depuis un fichier qui donne une description d'un DAG, **DPSO** qui implémente les heuristiques HEFT/DPSO (la version hybride de DPSO) et la version de base de DPSO, **HEFT** et **DHEFT** qui implémentent respectivement la version de base de HEFT et la variante utilisant la technique de duplication des tâches appelée Duplication-based HEFT (DHEFT).
2. La vue : c'est la partie qui gère l'interface graphique de l'application.

Le schéma dessus montre les classes C++ de notre application :

[1] GNU (**G**NU' **N**ot **U**nix) est un système d'exploitation libre lancé en 1983 par Richard Stallman, puis maintenu par le projet GNU. Il reprend les concepts et fonctionnement d'UNIX et permet l'utilisation de tous les logiciels libres et non seulement ceux réalisés dans le cadre de projet GNU.

[2] Win32 est un ensemble normalisé de fonctions qui permettent aux logiciels applicatifs de se servir des fonctionnalités du système d'exploitation de la famille Windows. Successeur de Win16, il a été introduit en 1993 dans les produits 32 bits de la Windows comme Windows NT, Windows 2000, et Windows 95.

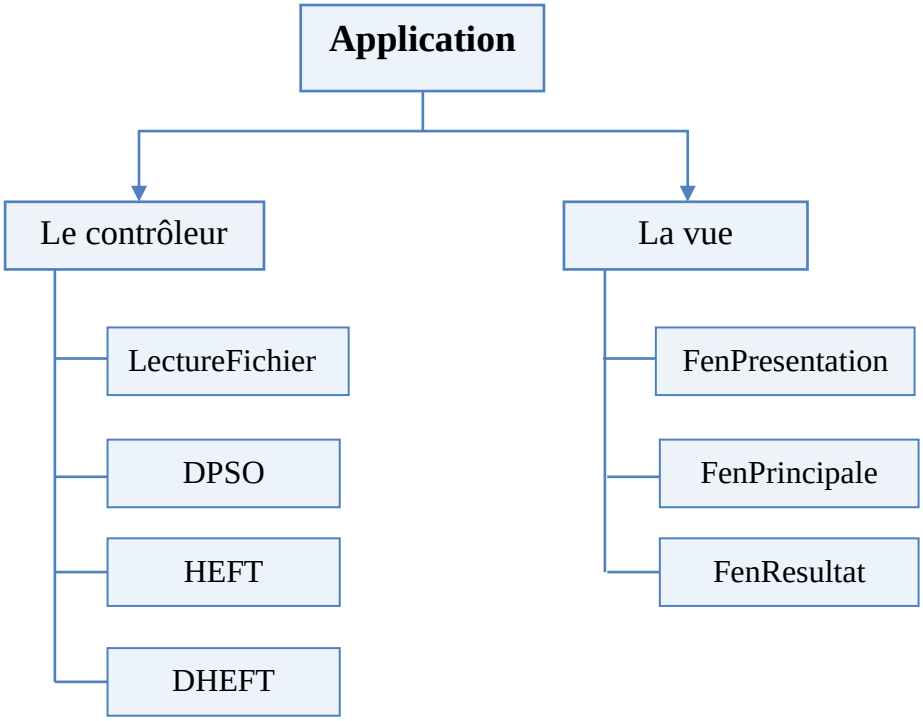


Figure 4.1 : Diagramme des classes.

2.3. Description de l'interface graphique

L'interface graphique se compose de :

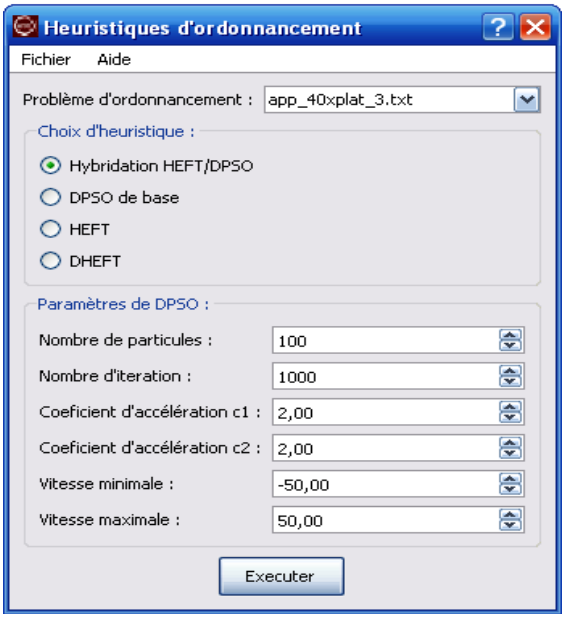
- **Splash screen :** pour faire patienter l'utilisateur au moment de démarrage de l'application

Figure 4.2 : Splash Screen de l'application.

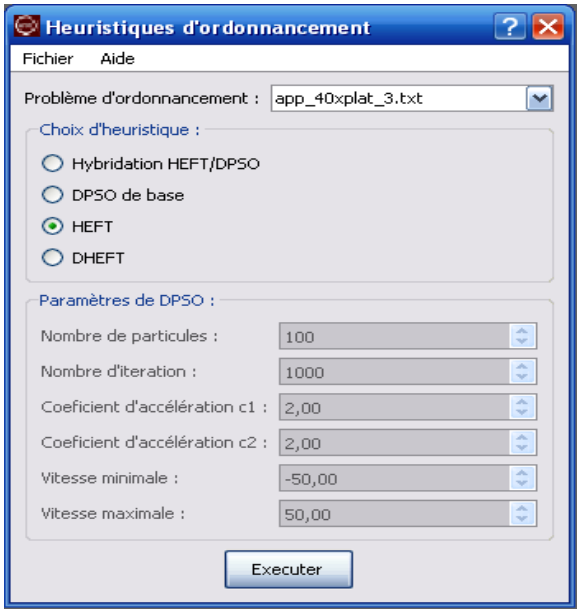
- **Fenêtre de présentation** : Elle présente des informations sur le nom, réalisateurs et l'encadreur du projet.

Figure 4.3 : Fenêtre de présentation.

- **Fenêtre principale** :



a. L'apparence de la fenêtre principale dans le cas où le choix est porté sur HEFT/DPSO.



b. L'apparence de la fenêtre principale dans le cas où le choix est porté sur HEFT.

Figure 4.4: Fenêtre principale.

Cette fenêtre est constituée de :

- une barre de menu ayant le menu Fichier et Aide. Le menu Aide affiche la fenêtre 'A propos' suivante :

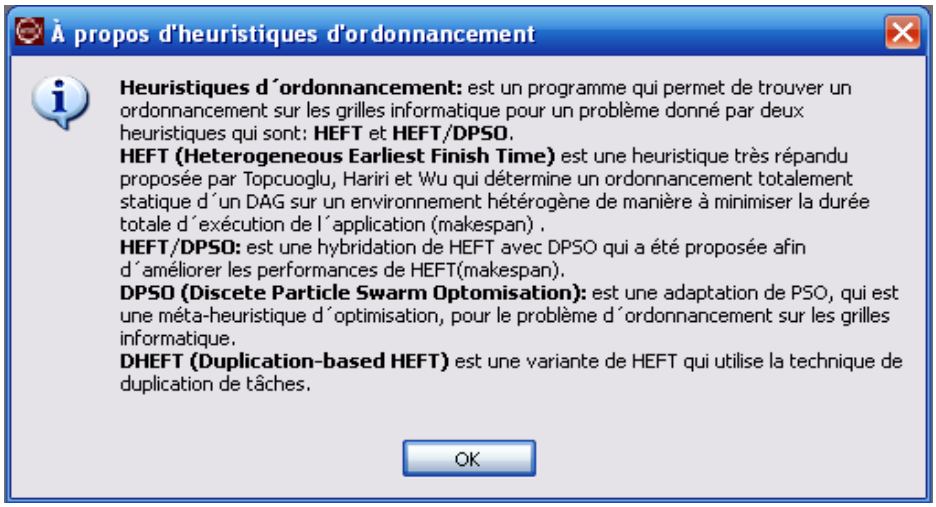


Figure 4.5 : La fenêtre 'A propos'.

- Une liste de problèmes d'ordonnancement décrits dans des fichiers spécifiques.
- Le choix d'heuristique : l'utilisateur a le choix entre HEFT/DPSO DPSO de base, HEFT ou DHEFT.
- Les champs pour le réglage des paramètres de DPSO qui ont des valeurs typiques par défaut [4.6]. Ces paramètres sont inaccessibles dans le cas où le choix d'heuristique est porté sur HEFT ou DHEFT (figure IV.4.b).

➢ **La fenêtre des résultats** : donne le résultat d'ordonnancement trouvé par l'heuristique choisie avec sont makespan (91 en unité de temps). La figure suivante donne le résultat d'ordonnancement donné par HEFT pour un problème de 11 tâches à exécuter sur une plate-forme de 3 machines. **AST** est le temps de début d'exécution de la tâche sur le processeur indiqué et **AFT** est le temps de fin d'exécution de la tâche.

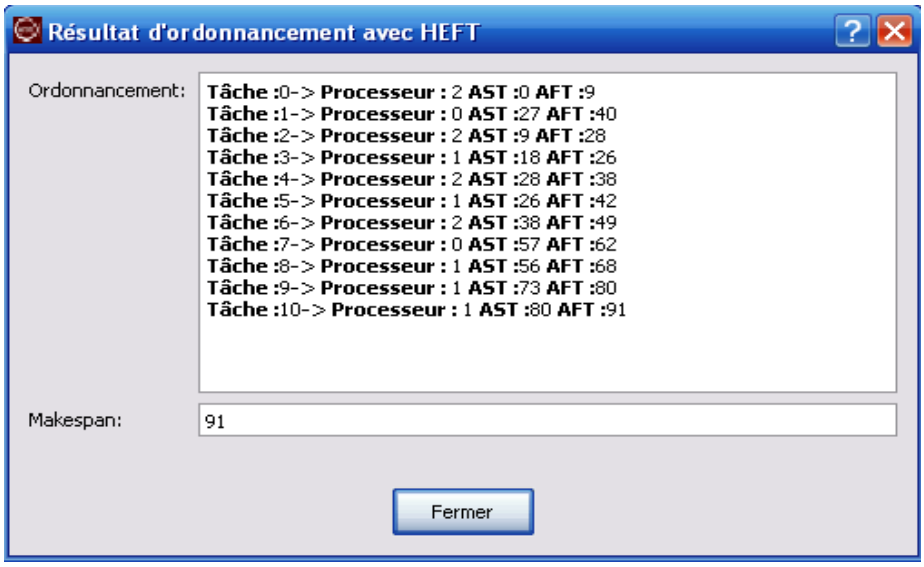


Figure 4.6: Résultat d'ordonnancement donné par l'heuristique HEFT.

- **Les messages d'erreurs :** Dans le cas où le fichier d'entrée n'a pas été trouvé dans le dossier 'Input' ou n'a pas le format spécifique décrit ci-dessous, des messages d'erreurs seront affichés pour avertir l'utilisateur.

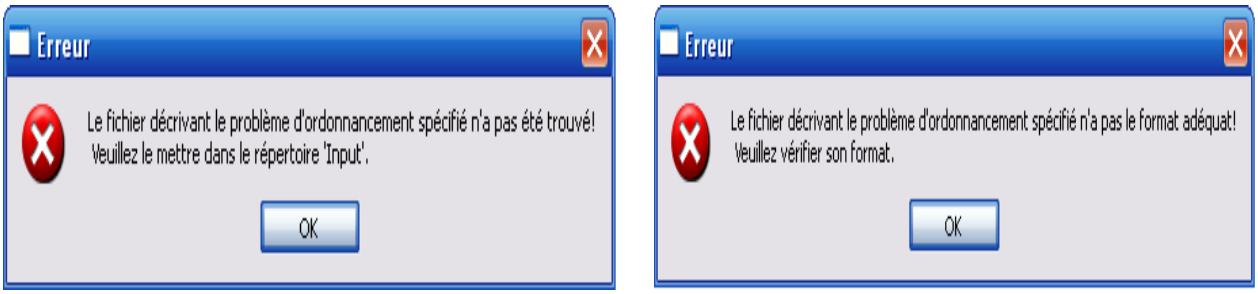


Figure 4.7: Les messages d'erreurs.

2.4. Le format des fichiers d'entrée

Les fichiers d'entrée contiennent les informations nécessaires pour la description des problèmes d'ordonnancement. Ils décrivent les DAGs (Direct Acyclic Graph) qui représentent l'application et les plateformes sur lesquelles s'exécutent ces applications. Nous avons utilisé un générateur de graphe de tâches pour crée ces fichiers. Ce programme est nommé **RandomTaskGRaphGenerator.** Il prend comme paramètre d'entré le nombre de tâche que le DAG doit avoir et le génère automatiquement en le mettant dans un fichier. Un fichier d'entrée a le format suivant :

1. Spécification du format de fichier : c'est une chaine de caractère spécifique (exemple 'motcle') utile pour les tests lors de l'ouverture et la lecture du fichier d'entrée par l'application.
2. Nombre de tâches qui forme l'application à ordonnancer.

3. Nombre de machines de la plateforme.
4. Matrice de coût de calcul ayant la dimension 'Nombre de tâches x Nombre de machines' (Les indices des lignes représentent les numéros des tâches et les indice des colonnes représentent les numéros des machines). L'unité des valeurs de cette matrice est l'unité du temps.
5. Matrice de vitesse de transfert de données entre les machines calcul ayant la dimension 'Nombre de machines x Nombre de machines'(les indices des lignes et des colonnes représentent les numéros des machine). La valeur 0 signifie qu'il n'y a pas de transfert entre une machine et une autre (aucun lien) aussi la vitesse de transfert entre une machine et elle-même est considérée comme nulle. Son unité est l'unité du temps sur celle du volume de données.
6. Matrice de données à transférer entre les tâches ayant la dimension 'Nombre de tâches x Nombre de tâches (les indices des lignes et des colonnes représentent les numéros des tâche). La valeur '-1' signifie qu'il n'y a pas de données à transférer entre les tâches. La valeur 0 veut dire qu'une tâche a une dépendance autre que le transfert de données avec une autre tâche. Et pour les valeurs positives, ça représente le volume de données à transférer entre les deux tâches. Son unité est celle du volume de données (l'octet par exemple).

La figure suivante montre un Exemple d'un fichier d'entrée décrivant un problème d'ordonnancement d'une application de 10 tâches sur une plateforme de 3 machines.

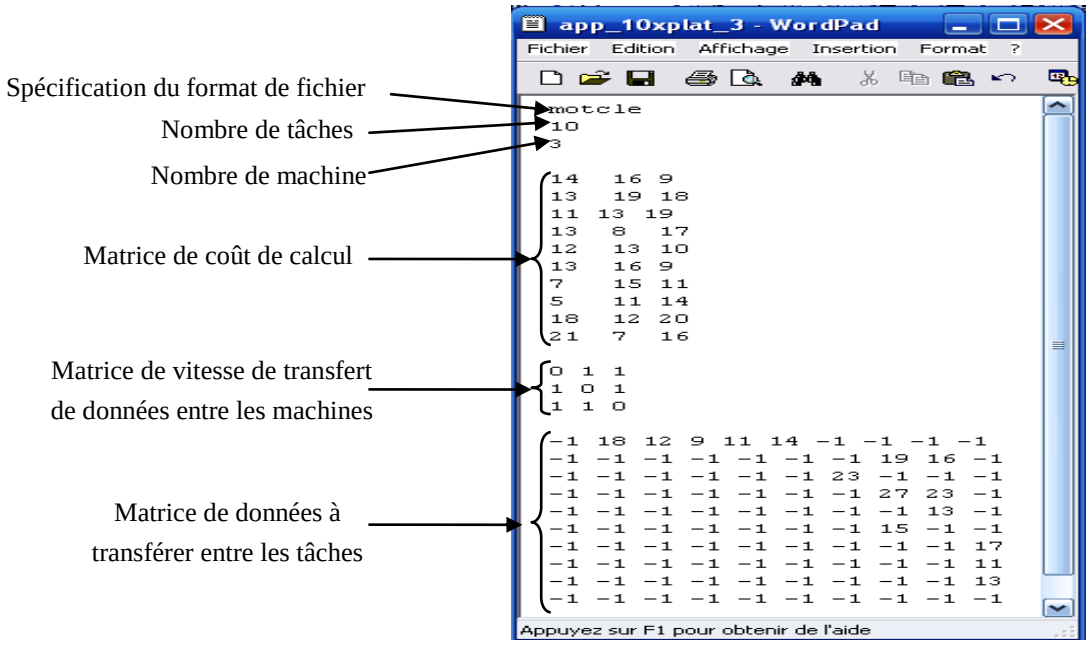


Figure 4.8: Exemple d'un fichier d'entrée.

3. Evaluation

Pour évaluer les performances de notre méthode 'Hybridation HEFT/DPSO', nous avons dû au premier lieu effectuer un choix portant sur les valeurs a affecté aux paramètres du DPSO et avons effectué une série de tests comparatifs entre :

- La méthode hybridé HEFT/DPSO.
- DPSO la version de base.
- l'heuristique HEFT.
- HEFT version duplication de tâches (DHEFT).

Nous avons réalisé ces tests sur six applications ayant un nombre différent de tâches (10, 20 40, 60, 80 et 100 tâches) qui s'exécuter sur des plateformes de 2, 3, 5, 7 ou 10 machines. Puisque les résultats de HEFT/DPSO et DPSO sont stochastiques, nous avons répété leur exécution dix fois et pris le meilleurs makespan trouvé.

3.1. Choix des valeurs des paramètres de DPSO

Nous avons fait varier les valeurs des paramètres de DPSO pour choisir les valeurs qui donne les meilleurs résultats en termes de makespan. Le tableau ci-dessous montre les valeurs typiques proposé dans [4.6] et le choix que nous avons retenu pour notre étude comparative.

	Choix 1 (Valeurs typique proposé dans [4.6])	**Choix 2 (Meilleur choix trouvé)**
Nombre de particules	50	50
Nombre d'itération	1000	1000
Coefficient c_1	2	2
Coefficient c_2	**2**	**1**
Vitesse minimale	-50	-50
Vitesse maximale	50	50

Tableau 4.1: Les valeurs des paramètres de DPSO pour deux choix différent.

Nous avons choisi de réduire l'effet de la composante sociale (c_2 prend la valeur 1 au lieu de 2) pour permettre une plus grande exploitation de l'espace de recherche. Pour les autre paramètres, nous avons laissé leurs valeurs telle que proposé dans [4.6]. Pour le nombre de particule et d'itération, le choix est un compromis entre exploitation de l'espace de recherche et vitesse d'exécution de DPSO.

Pour comparer l'influence des choix des paramètres de DPSO, nous avons effectué des tests en utilisant la version de base de DPSO (initialisation aléatoire des particules) sur des applications ayant différents nombre de tâches sur une plateforme de 10 machines. Les résultats sont donnés dans le tableau suivant :

Nombre de tâche	Choix 1(Valeurs typique proposé dans [4.6])	Choix 2 (Meilleur choix trouvé
10	91	91
20	171	165
40	354	335
60	432	388
80	454	384
100	705	665

Tableau 4.2: Makespan donné par DPSO en fonction de nombre de tâches avec deux choix de paramètres de DPSO.

La courbe suivante traduit les résultats du précédent tableau :

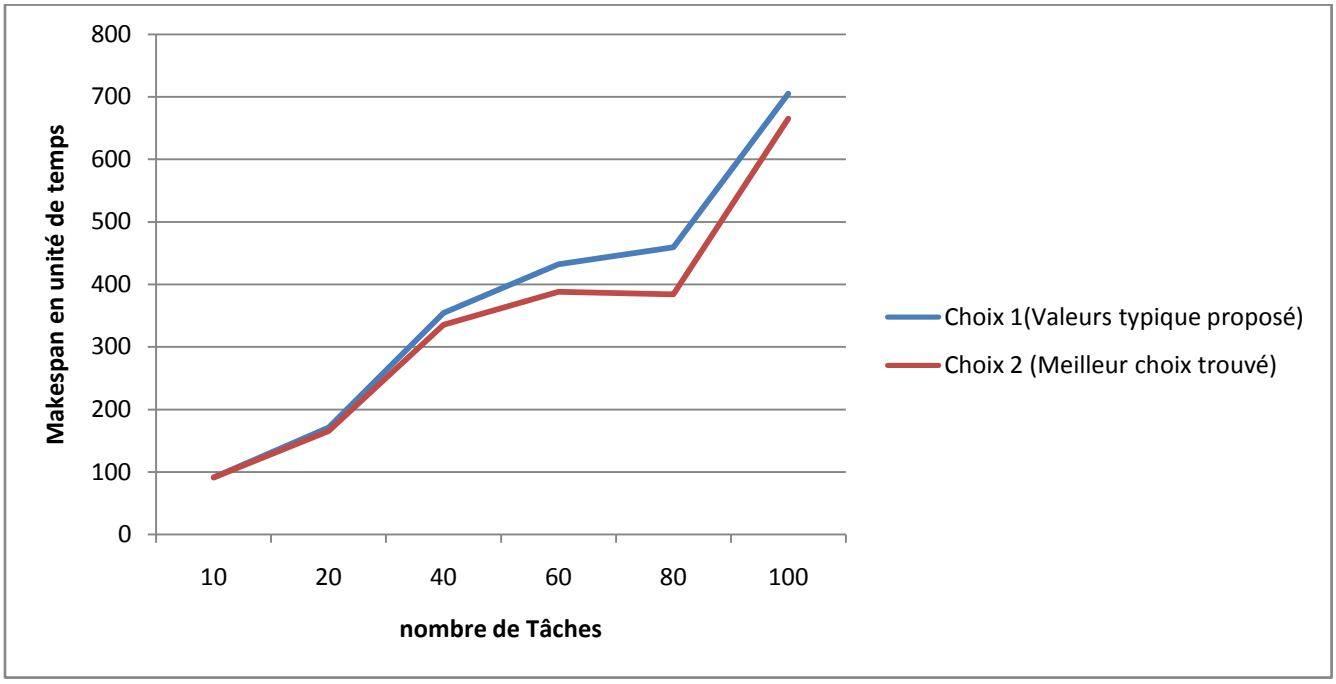


Figure 4.9: Courbe représentative de Makespan donné par DPSO en fonction de nombre de tâches avec deux choix de paramètres de DPSO.

Nous constatons que le makespan obtenu par l'exécution de DPSO avec les valeurs des paramètres du choix 2 est toujours meilleur que celui trouvé avec ceux de choix 1.

3.2. Etude comparative

Dans ce qui suit, nous avons retenu les valeurs du choix 2 pour les paramètres de DPSO.

Paramètres de DPSO	Valeur du paramètre
Nombre de particule	50
Nombre d'itération	1000
Coefficient c_1	2
Coefficient c_2	**1**
Vitesse minimale	-50
Vitesse maximale	50

Tableau 4.3: Les valeurs des paramètres de DPSO utilisé dans l'étude comparative.

3.2.1. Makespan en fonction de nombre de machine

Nous avons pris une application de 40 tâches qui s'exécutent sur six plates-formes différentes ayant : 2, 3, 5, 7, et 10 machines. Les résultats sont donnés dans le tableau suivant

Nombre de machine	HEFT/DPSO	DPSO	HEFT	DHEFT
2	733	736	922	728
3	560	580	620	549
5	425	437	441	437
7	383	392	399	398
10	296	329	308	308

Tableau 4.4: Makespan donné par quatre heuristiques en fonction de nombre de machine.

La courbe suivante traduit les résultats du précédent tableau :

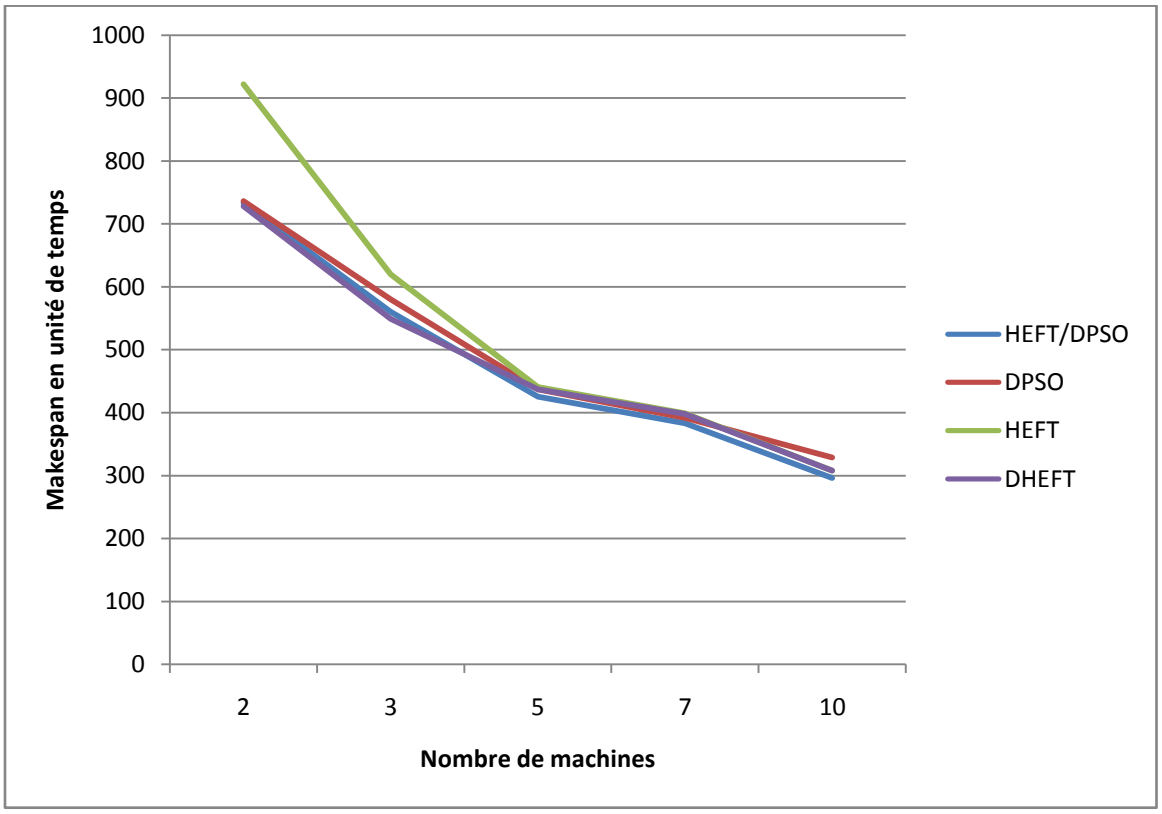


Figure 4.10: Courbe représentative de Makespan donné par quatre heuristiques en fonction de nombre de machine.

En analysant la courbe, nous constatant que : le makespan diminue pour les quatre algorithmes d'ordonnancement en augmentant le nombre de machine dans la plateforme dû à la distribution des tâches sur l'ensemble des machines. Le makespan de notre méthode hybridée HEFT/DPSO est toujours plus petit que celui de HEFT et de DPSO. Le makespan de HEFT/DPSO est, de façon général, inferieur à celui de DHEFT sauf pour une plateforme constituée de trois machines.

3.2.2. Makespan en fonction de nombre de tâches d'application

Nous avons pris des applications de 10, 20, 40, 60, 80 et 100 tâches s'exécutant sur une plateforme de 5 machines.

Nombre de tâche	HEFT/DPSO	DPSO	HEFT	DHEFT
10	71	71	84	62
20	225	228	299	270
40	431	411	441	437
60	446	511	654	591
80	434	573	523	395
100	681	849	701	636

Tableau 4.5: Makespan donné par quatre heuristiques en fonction de nombre de tâches.

La courbe suivante traduit les résultats du précédent tableau :

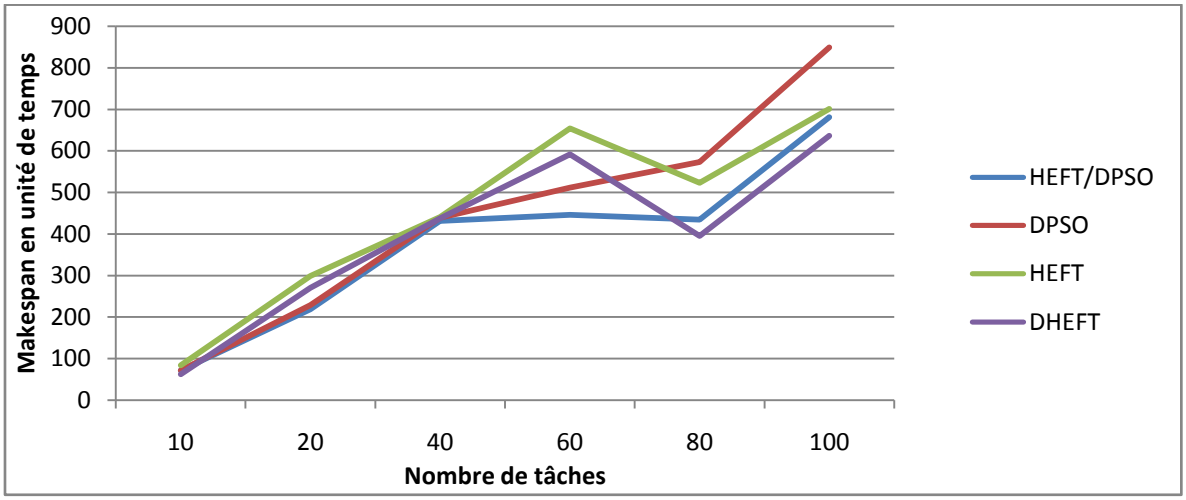


Figure 4.11: Courbe représentative de Makespan donné par quatre heuristiques en fonction de nombre de tâches.

En analysant la courbe, nous constatant que : le makespan augmente en augmentant le nombre de tâches et ça pour les quatre algorithmes d'ordonnancement. Nous remarquons aussi que le makespan de notre méthode hybridée HEFT/DPSO est toujours plus petit que celui de HEFT et de DPSO. Et pour HEFT_dup, son makespan est proche de celui de HEFT/DPSO mais supérieur pour les applications avec 10, 20 et 40 tâches et cette différence est plus marquée pour l'application ayant 60 tâches. Cependant, pour les applications avec 80 et 100 tâches, le makespan de DHEFT est inférieur par rapport à HEFT/DPSO dû à l'effet de la duplication des tâches dans DHEFT qui minimise le temps de communication entre les machines c.à.d. le temps de transfert des données entre les tâches qui s'exécutent sur des machines différentes.

3.2.3. Comparaison avec plateforme Homogène et Hétérogène

Nous avons exécuté une application de 60 tâches sur une plate-forme de sept machines Homogène/Hétérogène

Plate-forme	HEFT/DPSO	DPSO	HEFT	DHEFT
Homogène	837	842	1011	924
Hétérogène	350	421	483	479

Tableau 4.6: Makespan donné par quatre heuristiques pour une plateforme homogène et une autre hétérogène.

La courbe suivante traduit les résultats du précédent tableau :

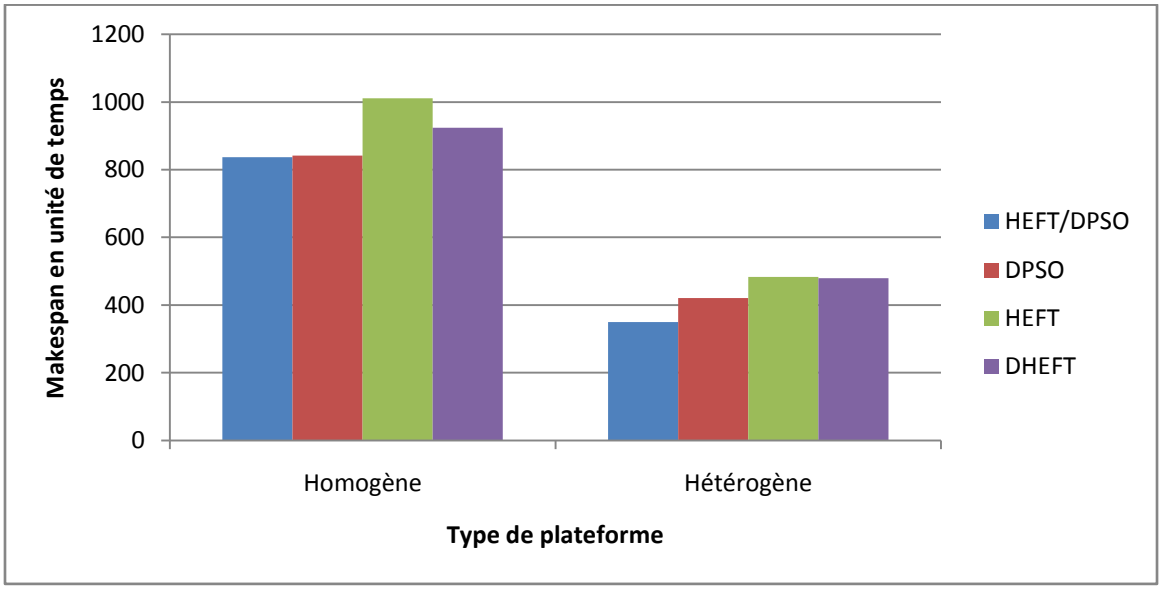


Figure 4.12: Histogramme représentatif de Makespan donné par quatre heuristiques pour une plateforme homogène et une autre hétérogène.

3.2.4. Synthèse

En comparant les différents makespans des applications sur différentes plates-formes, nous constatons que les makespans de la méthode hybridée HEFT/DPSO sont toujours plus petits par rapport à ceux obtenus avec HEFT et DPSO de base, et plus au moins proches de ceux de DHEFT

4. Conclusion

Dans ce chapitre, nous avons présenté l'implémentation de notre application et avons effectué une comparaison avec d'autres heuristiques dans le but d'évaluer notre travail. Nous avons constaté une amélioration dans les résultats (makespans) obtenus par la méthode hybridée HEFT/DPSO par rapport à HEFT et DPSO de base et des résultats plus au moins proches de ceux donnés par la DHEFT.

L'objectif de notre travail est d'améliorer le makespan des applications exécutées sur les grilles de calcul. Nous nous sommes intéressés au problème d'ordonnancement de tâches sur les grilles de calcul.

Après avoir étudié les algorithmes et les heuristiques utilisés pour résoudre ce problème, nous nous sommes intéressés aux deux algorithmes qualifiés d'intéressant dans la littérature, à savoir l'algorithme HEFT et la métaheuristique PSO.

Nous avons conçu et implémenté une heuristique hybride basée sur HEFT et une version de PSO adaptée aux problèmes d'ordonnancement dans les grilles de calcul.

Pour évaluer les performances de la méthode que nous avons implémentée, une étude comparative a été réalisée avec d'autres algorithmes d'ordonnancement qui sont DPSO, HEFT et DHEFT.

Cette étude qu'on a menée dans ce travail montre que le makespan de la méthode que nous avons proposée est meilleur que les autres. En effet, les makespans de la méthode hybride HEFT/DPSO obtenus pour différentes applications s'exécutant sur différente plateformes sont toujours plus petits par rapport à ceux obtenus avec HEFT et DPSO de base, et plus au moins proches de ceux de DHEFT.

Ce travail nous a permis de nous façonner une idée sur les grilles de calcul, et l'ordonnancement des tâches sur ce types de plate-forme.

Comme perspectives à notre travail :

- Utilisation de HEFT version duplication de tâches avec PSO adapté pour la duplication des tâches.
- Utilisation d'une technique comme les algorithmes génétique pour regrouper les tâches ayant beaucoup de données à transférer enter elles (approche de clustering) pour former des méga-tâche, ensuite ordonnancer ces méga-tâches en utilisant HEFT.
- Prendre en considération l'hétérogénéité des liens de communication entre les machines (matrice de vitesse de transferts de données).

[1.1]: T. MONTEIL : Du cluster à la grille sous l'angle de la performance, mémoire de l'Habilitation à Diriger des Recherches de l'Institut National polytechnique de Toulouse, 2010.

[1.2]: Wikipédia.

[1.3]: R. BUYYA and M. CHETTY: Weaving Computational Grids: How Analogous Are They With Electrical Grids? , 2002.

[1.4]: J. E. GHAFFOUR: Sécurité Dans Les Grilles De Calcul, Campus Universitaire de Beaulieu 35042 RENNES Cedex – France, 2004.

[1.5]: F. TOUAHIR et M. OULD FELLA : Ordonnancement de Tâches Dépendantes sur la Grille de Calcul, 2013.

[1.6]: V. BERSTIS: Fundamentals of Grid Computing, 2002.

[1.7]: T. CHARRON : Le projet européen de grillede calcul (DataGrid) : Concept de grilles de calcul. Objectifs du projet et état d'avancement. Conservatoire National des Arts et Métiers Centre d'enseignement de Lyon - Mai 2003.

[1.8]: J. YU, R. BUYYA and K. RAMAMOHANARAO: Workfow Scheduling Algorithms for Grid
Computing, Grid Computing and Distributed Systems (GRIDS) LaboratoryDepartment of Computer Science and Software EngineeringThe University of Melbourne, VIC 3010 Austraila, 2008.

[1.9]: L. CANTONE et R. UNAL : Le Grid Computing et son utilisation dans les entreprises et les industries, UNIVERSITE Claude Bernard, Lyon 1.

[1.10]: S. DIAKITE : Contribution à l'ordonnancement de lots de graphes de tâches sur une plate-forme hétérogène, thèse pour obtenir le grade de docteur de l'université de FERANCHE-COMTE, 2010.

[1.11]: O. HAMDI-LARBI : Étude de la Distribution, sur Système à Grande Échelle, de Calcul Numérique Traitant des Matrices Creuses Compressées, thèse pour obtenir le grade de docteur en informatique, 2010.

[1.12]: M. A. ALOULOU: Introduction aux problèmes d'ordonnancement, LAMSADE Université Paris Dauphine, 2005.

[1.13]: H. SABBAH : Modélisation et dimensionnement d'une plate-forme hétérogène de service, thèse pour obtenir le grade de docteur de l'université de FERANCHE-COMTE, 2009.

[1.14]: M. AMIGUET : Recherche heuristique et méta-heuristique, Haute école d'ingénierie, 2008-2009.
[1.15]: R. SAKELLARIOU and H. ZHAO: A Hybrid Heuristic for DAG Scheduling on Heterogeneous Systems, Department of Computer Science University of Manchester Oxford Road Manchester M13 9PL, U.K.

[1.16]: T. N'takpé : Ordonnancement de tâches parallèles sur plates-formes hétérogènes Partagées, thèse pour le grade de docteur de l'université Henri Poincaré, Nancy 1, 2009.

[1.17]: D. CULLER et al.: LogP: Towards a Realistic Model of Parallel Computation, Computer Science Division, University of California, Berkeley, 1993.

[1.18]: B. MIEGEMOLLE : Prédiction de comportement d'applications parallèles et placement à l'aide de modèles économiques sur une grille de calcul, thèse pour obtenir le grade de docteur de l'université de Toulouse, 2008.

[1.19]: S. TISON, Graphes : Terminologie, Représentation, UFR IEEA 2005-2006.

[2.1]: Y. COOREN : Perfectionnement d'un algorithme adaptatif, d'Optimisation par Essaim Particulaire. Applications en génie médical et en électronique, thèse pour obtenir le grade de docteur de l'université Paris 12 Val De Marne, 2008.

[2.2]: A. TOUMI : Restauration Adaptative d'Image par les Méthodes Intelligentes, thèse pour obtenir le grade de docteur en sciences spécialité : Génie Electrique, 2013.

[2.3]: A. EL DOR : Perfectionnement des algorithmes d'Optimisation par Essaim Particulaire. Applications en segmentation d'images et en électronique. Thèse pour obtenir le grade de docteur en informatique de l'université PARIS-EST, 2012.

[3.1]: H. Amrouche : L'hybridation des métaheuristiques, mémoire de magister en Automatique, UMMTO.2012.

[3.2]: E. Talbi : A taxonomy of hybrid metaheuristics. Journal of Heuristics, tome 8, pages 541-694, 2002.

[3.3]: F. Glover: Heuristics for integer programming using surrogate constraints. Decision Sciences, tome 8, n° 1, pages 156-166, 1977.

[3.4]: J.J. Grefenstette: Incorporating problem specific knowledge into genetic algorithms, in Genetic Algorithms and Simulated Annealing, Davis, L. D. (Ed.), London: Pitman. (1987).

[3.5]: H. Mühlenbein, M. Gorges-Schleuter et O.Kramer: Evolution Algorithms in Combinatorial Optimization. Parallel Computing, tome 7, pages 65-88, 1988.

[3.6]: H.M. Voigt, J. Born, and I. Santibanez-Koref: Modelling and simulation of distributerd evolutionary search processes for function optimization. Parallel Problem Solving from Nature, 496: 373-380, 1990. LNCS.

[3.7]: Puchinger, J. et G.R. Raidi : Combining Metaheuristics and Exact Algorithms in Combinatorial Optimization: A Survey and Classification, in Proceedings of the First International Work-Conference on the Interplay Between Natural and Artificial Computation, 2005, Springer (Ed), Las Palmas, Spain, LNCS, pp.41-53.

[3.8]: Haluk Topcuoglu, Salim Hariri and Min-You Wu: Performance-Effective and Low-Coplexity Task Scheduling for Heterogeneous Computing.

[3.9]: Y. Zhang, Y. Inoguchi, and H. Shen: A dynamic Task Sceduling Algorithm for Grid Computing System.

[3.10]: Hesam Izakian, Behrouz Tork Ladani, Ajith Abraham and V´aclav Sn´aˇsel: A Discrete Particule Swarm Optimization Aproach for Grid Job Scheduling.(2010)

[4.1]: www.depannetonpc.net/lexique/lire_74_langage-c.html

[4.2]: www.depannetonpc.net/lexique/lire_programmation-orientee-objet.html

[4.3]:http://download.qt-project.org/snapshots/qt/4.8/2014-03-17_517/

[4.4]: https://docs.google.com/file/d/0B1LZKOBK6wYBQXl1d2pEX3FnZlE/edit

[4.5]: http://fr.openclassrooms.com/informatique/cours/compiler-facilement-qt .

[4.6]: http://www.swarmintelligence.org/tutorials.php .